TURING 图灵程序设计丛书

SAMS

- 久负盛名的C语言神奇入门书
- 来吧，现在就开始攻克C语言
- 从这里，爱上编程



写给大家看的C语言书（第2版）

[美] Greg Perry 著

谢晓钢 刘艳娟 译

人民邮电出版社
POSTS & TELECOM PRESS



写给大家看的C语言书（第2版）

我在大学教了多年的编程课，遗憾的是，大多数教材对于初学者来说都太厚太难理解了。而本书则完全从初学者的角度来写，我强烈推荐本书给那些一直畏惧编程的学生。

—— Bruce Conklin，Phoenix 大学教授

这本书带我迅速入门，它让我从此爱上了编程。

—— Eric M.，13 岁

没有编程基础，被 C 语言的术语和细节搞得晕头转向，手头的教科书无数次拿起又放下、总是读不进去，甚至已经开始考虑放弃？不要灰心！本书将带你开始一次轻松有趣的 C 语言之旅，并且使你在旅途的终点自信满满地迎接憧憬无限的明天。

这是一部别开生面、与众不同的 C 语言经典入门著作，在全球畅销已经近 20 年，重印数十次。书中，作者好像坐在你身边的老朋友，拉着家常，将 C 语言的基础知识、必备的实战技能和宝贵编程经验一一道来。没有云山雾罩，没有前因后果的行话，没有艰深而且不必要的内部技术细节，没有沉闷晦涩的长篇大论，只有生动的漫画、丰富的图示、幽默通俗的文字、细致的分析阐述、引人入胜的趣闻轶事和短小精悍的章节安排。作者不仅将重要的知识点以“线索”、“警告”、“陷阱”、“难点”、“小结”等活泼的形式呈现给读者，而且还提供了一个贯穿全书的 21 点游戏程序，完美地将旅途上所有的景点巧妙地嵌在其中。

本书不仅已经成功地使无数没有编程经验的初学者包括许多十几岁的孩子掌握了 C 语言，也是全球许多著名高校的程序设计课程教材。

SAMS

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010)51095186
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机/程序设计/C

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-21635-9

ISBN 978-7-115-21635-9

定价：39.00 元

前　言

眼看着你的朋友们一个个得到C语言编程的工作而你只能被冷落在一边，你是否有些心灰意冷？你是否想学习C语言却苦于没有足够的精力？你的老旧的、磨损严重的计算机是否需要用一种热门的编程语言来提高运行速度？这本书正是医生为你对症下药开出的处方。

本书不像一般的计算机图书那样居高临下地说教，它站在读者的角度讲解，通俗易懂。本书就像是你最好的朋友坐在身边教你C语言。作者使用平实的语言而非“术语”。与其他书相比，本书中那些短小的章节、丰富的图示以及偶尔出现的幽默直白的讨论，能指导你更迅速、更友好、更容易地穿过C语言编程的迷宫。

读者对象

这是一本入门书。如果你从来没有写过程序，那么这本书正好适合你。学习本书不需要任何编程概念的知识。就算你不能拼写C，也可以通过本书学会用C语言编程。

短语*绝对初学者*[①]（absolute beginner）在不同情况下有不同的意思。你以前可能尝试过学习C语言但又放弃了。许多书和培训班都给C语言强加了太多的技巧。你可能曾经用其他语言编写过程序但是刚开始学习C语言。如果是这样，请读下去，学完32章之后，你将了解C语言。

本书特色

本书不使用那些初学者不需要的内部技术操作来云山雾罩地分析问题。作者（我）坚信应该细嚼慢咽地学好基本原理。一旦你打好了基础，“更难”的部分看起

① 原书名为《C语言绝对初学者指南》。——编者注

来也就没那么难了。本书教给你真正的C语言入门知识。

也可以认为C语言是一种极其神秘且难学的语言。许多人不止一次地尝试学习C语言。其实问题很简单：任何主题，不管是脑外科、邮件分类还是C语言编程，只要解释清楚了，都很容易。谁都无法教会你什么，因为只有你能教会你自己。但是如果教师、教科书或教学视频没有令主题变得简单且有趣，你就不会想要学下去了。

你可以尝试寻找比本书更易懂的学习C语言的方法。如果你找到了，请告诉我，因为我也想阅读它。（你一定在想可能我会把买书的钱退还给你？）说真的，我想给你提供一种你在其他地方找不到的帮助。

本书最大的优势在于作者（还是我）真的喜欢编写C语言程序，并且更喜欢好为人师地教人学C语言。我相信你也一定能喜欢上C语言。

本书的设计元素

与许多计算机类图书一样，本书包含了很多有用的提示、技巧及警告等。你将会看到很多图标（小图片），吸引你去注意这些内容。瞄一眼图标就能知道它旁边文本的作用。下面介绍书中的图标。



警告 此图标指出在所讨论的主题中可能会遇到的问题。通常，它指明一个需要留意的警告，或者是可能发生的问题的解决方法。



线索 本书的很多提示和线索（它们有很多）都由此图标来强调。当某个精彩的特性或编码技巧切合你正在阅读的主题时，这个图标指明你可以如何加以利用。



说明 在C语言中，某些主题应进行更深层次的理解。此图标告诉你一些以前可能没有想到过的东西，比如正在讨论的主题还有一种新用法。



请跳过这里，这有点儿难 如果你只想了解初学者应该学习的内容，就不要阅读本框中的内容。实际上，你也许爱看这些内容，但是如果你跳过不看，也不会影响你理解本章内容。

有时候你会在页边看到**趣闻轶事**。这些**趣闻轶事**传递了关于计算机、编程和C语言的一些有意思的信息。

每一章最后都回顾这一章中需要记住的知识要点。在**奖励**和**陷阱**标题下的内容分别列出了你该做和不该做的事情。**小结**一节是本书的一大特色，它将所有内容串了起来。这一节描述了这一章的主要目的，给出一个代码示例强调所教的概念，并对此示例代码进行讲解分析。你会发现每一章小结（从第2章开始）都是对本章要点的一个很好的总结。

本书使用如下的版式约定：

- 代码行、变量和在屏幕上看到的任何文本用**等宽字体**；
- 格式行上的占位符用*斜等宽字体*；
- 新术语用*楷体*；
- 语法解释中的可选参数用方括号（[]）括起来，使用这些参数时，不需要输入方括号。

寓学于乐

附录B给出了一个完整可用的21点扑克牌游戏程序。此程序尽可能地短，同时又不牺牲代码的可读性和游戏功能。该游戏具有一般性，可在所有的C编译器上运行。因此，你看不到迷人的图形，但是一旦你学会了C语言，就可以很容易地使用编译器特定的图形、声音和数据输入程序来改进该程序。

此程序尽可能多地使用了本书的内容。本书所教的几乎所有主题都出现在了21点游戏程序中。其他太多的书仅仅提供一些代码片段，而这个游戏让你看到了全景。在学习本书的过程中，你会越来越深入地理解这个游戏。

现在该做什么

翻到下一页开始学习C语言吧！





书号：978-7-115-19432-9

你必须知道的495个C语言问题

●与《代码大全》齐名的经典著作
●揭示微软成功的技术奥秘
●C语言高手的秘籍

内容简介

编写高质量的、没有 bug 的程序，是每位程序员所追求的目标。但随着软件规模越来越大，功能日趋复杂，这一目标变得越来越困难。

本书揭示了微软公司应对质量挑战、开发出世界级代码的技术内幕，作者在自己不断探索、实践和思考的基础上，系统总结了多年来指导微软各团队的经验，将其凝聚为许多切实可行的编程实践指导，可谓字字珠玑。正因如此，本书被公认为与《代码大全》齐名的编程技术名著，曾于 1993 年荣获有软件开发奥斯卡奖之称的 Jolt 生产效率大奖。书中内容主要针对 C 语言，但其中的思想对目前的各主流语言编程也完全适用。

书评

“本书所蕴含的思想精髓可以说是不朽的，它彻底改变了我的编程方式。”

——David Kline，微软资深工程师，
.NET Compact Framework 核心开发成员

“如果你想成为真正的 C 程序员，如果你想打造真正优秀的软件开发团队，请读本书。不要找什么借口去忽视它，是十足的愚蠢之举，蜂拥而至的 bug 会很快来惩罚你。”

——ACCU（C/C++ 用户协会）网站



书号：978-7-115-17286-0

C标准库

● C标准库“圣经”
● 提供完整源代码，全面深入阐述库函数的实现与运用
● C程序员必备参考书

内容简介

本书是由世界级 C 语言专家编写的 C 标准库经典著作。英文版已经重印十多次，影响了几代程序员。

本书结合 C 标准的相关部分，精辟地讲述了每一个库函数的使用方法和实现细节，而这正是一个真正的 C 程序员所必须掌握的。更重要的是，书中给出了实现和测试这些函数的完整源代码，可以让你更深入地学习 C 语言。不仅如此，本书还讨论了一些即使是最有经验的 C 程序员通常也不熟悉的知识，比如国际化和独立于区域设置的程序的编写、与构建库相关的概念和设计思想。

书评

“绝对一流的著作，C 程序员的饕餮大餐！”

——C/C++ 用户协会



书号：978-7-115-16427-8

C语言详解（第5版）

内容简介

本书是C语言经典教程，历经多次改版，为世界各国众多大学采用，迄今为止全球已有近10万读者通过本书掌握了C语言。

本书的最大特色是程序设计思想的介绍和C语言语法的讲述并重，不仅细致入微地讲述C语言知识，而且更多地强调学生解决实际问题和动手编程的能力。与国内众多C语言教材相比，书中突出了良好的程序风格和软件工程概念的重要性，并力求使学生比较早地建立软件开发的大局观。书中将教学难点——指针分散在多个章节分别讨论，使学生能够从实际运用中逐渐吸收指针用法的精髓，充分体现了作者丰富的教学经验。此外，本书的内容也非常新颖，讲授了用抽象管理复杂性、多进程和多线程、动态数据结构等知识。大量的案例、示例、常见编程错误、习题和程序设计项目，使本书成为课堂教学和自学参考皆宜的理想学习工具。

书评

“我连续几个学期使用本书作为教材，教授程序设计课程。学生都认为它可读性很强。……我认为本书极好地介绍了如何编程以及如何使用C语言。……我强烈推荐这本教材，无论是课堂教学还是自学。”

——Dan Grecu，伍斯特理工学院

“如果你想学习编程，用C作为实现语言，那么本书将是最好的选择之一。”

——Francis Glassborow，ACCU（C/C++用户协会）前主席



书号：978-7-115-16707-1

C语言程序设计：现代方法

内容简介

本书是C语言领域的经典之作，已被全球200多所学校采用为教材，包括哈佛大学、麻省理工学院、斯坦福大学、加州大学伯克利分校、耶鲁大学、加州理工学院等诸多名校。

与一般C语言图书相比，本书的特色包括如下几点。

- 强调软件工程和现代编程理念，在知识的阐述中突出工业界的最佳实践、实际经验和编程风格，使学生能够合理运用所学，编写出可读性好、可靠性高和容易维护的代码。
- 注重实践，将C语言置于实际开发的大背景中，讲解应该何时和如何正确使用各个语言特性。
- 注重教学效果，由易而难、循序渐进、螺旋式地讲述C语言，很好地处理了指针和位运算等教学难点。
- 涵盖全面，讨论了标准C和C标准库的全部特性，包括信号、setjmp/longjmp和可变长参数列表等其他书中很少涉及的内容。
- 精心选择了丰富的实例、习题，贴近实战，与叙述文字相得益彰。
- 每章末尾都有生动的常见问题解答，更加丰富了本书内容。

这些使本书成为C语言学习的理想教材，被誉为“近10年来最好的一部C语言著作”。

书评

“本书为C语言提供了非常完整、清晰的概述，可以说它比我遇到的任何C语言书籍都要好……作者对编程风格和代码规范的处理非常出色……这本书对各种层次读者而言都极为理想。”

——Henry H. Leitner，哈佛大学

“出色的组织性、可读性和全面性……使用本书的学生应该会迅速对编写真正的C程序充满信心。”

——Samuel P. Harbison，C++标准委员会前主席，C: A Reference Manual一书作者

“我们以前曾经使用Kernighan & Ritchie的经典著作以及许多其他同类图书作为教材。但是当我们看到本书后，立刻做出了更换教材的决定。事实证明，所有人都非常满意。学生喜爱它的清晰易读，老师则欣赏书中对C程序设计现代风格的强调。”

——Orjan Ekeberg，瑞典皇家理工学院

目　录

第一部分　C语言初步

第1章　初识C语言 ····· 3

1.1　什么是程序 ····· 4
1.2　编写C程序需要什么 ····· 5
1.3　编程过程 ····· 6
1.4　使用C语言 ····· 7

第2章　从何处入手 ····· 11

2.1　概述 ····· 12
2.2　main()函数 ····· 13
2.3　数据类型 ····· 14
2.3.1　字符 ····· 14
2.3.2　数字 ····· 16
2.4　小结 ····· 18

第3章　代码注释 ····· 21

3.1　给代码加注释 ····· 22
3.2　注释详述 ····· 23
3.3　留白艺术 ····· 25
3.4　另一种注释方式 ····· 26
3.5　小结 ····· 27

第4章　输出 ····· 29

4.1　printf()做了什么 ····· 30
4.2　printf()的格式 ····· 30
4.3　打印字符串 ····· 31
4.4　转义序列 ····· 32
4.5　转化字符 ····· 33
4.6　小结 ····· 35

第5章　变量 ····· 39

5.1　变量类型 ····· 40
5.2　变量的命名 ····· 41
5.3　定义变量 ····· 41
5.4　在变量中存储数据 ····· 42
5.5　小结 ····· 44

第6章　字符串 ····· 47

6.1　字符串结束符 ····· 48
6.2　字符串的长度 ····· 49
6.3　字符数组：字符的列表 ····· 49
6.4　初始化字符串 ····· 52
6.5　小结 ····· 53

第7章　#include和#define ····· 55

7.1　包含文件 ····· 56
7.2　在哪里放置#include指令 ····· 58
7.3　定义常量 ····· 59
7.4　小结 ····· 61

第8章　输入 ····· 63

8.1　简述scanf() ····· 64
8.2　与printf()一起使用 ····· 65
8.3　使用scanf()的问题 ····· 65
8.4　小结 ····· 68

第9章　C怎么做数学运算 ····· 71

9.1　基本知识 ····· 72
9.2　运算符的优先级 ····· 74
9.3　用括号打破规则 ····· 75
9.4　多重赋值 ····· 75
9.5　小结 ····· 77

第二部分 操作空间

第 10 章 表达式还能用来做什么 …… 81
10.1 复合赋值 …… 82
10.2 小心优先级 …… 84
10.3 强制类型转换 …… 85
10.4 小结 …… 86
第 11 章 关系运算符 …… 89
11.1 测试数据 …… 90
11.2 使用if语句 …… 91
11.3 否则……：用else语句 …… 92
11.4 小结 …… 94
第 12 章 逻辑运算符 …… 97
12.1 获取逻辑 …… 98
12.2 逻辑运算符的优先级 …… 103
12.3 小结 …… 104
第 13 章 更高级的运算符 …… 107
13.1 条件运算符 …… 108
13.2 运算符++和-- …… 110
13.3 运算符sizeof() …… 111
13.4 小结 …… 112

第三部分 保持控制

第 14 章 循环 …… 117
14.1 while循环 …… 118
14.2 使用while语句 …… 119
14.3 使用do-while语句 …… 120
14.4 小结 …… 122
第 15 章 其他循环 …… 125
15.1 为了重复 …… 126
15.2 使用for循环 …… 128
15.3 小结 …… 130
第 16 章 终止循环 …… 133
16.1 使用break语句 …… 134
16.2 使用continue语句 …… 135
16.3 小结 …… 137
第 17 章 测试多个值 …… 141
17.1 使用switch语句 …… 142
17.2 switch语句中使用break语句 …… 144
17.3 妙用case语句 …… 144
17.4 小结 …… 145
第 18 章 输入和输出 …… 149
18.1 putchar()和getchar()函数 …… 150
18.2 关于换行符的思考 …… 152
18.3 getch()函数 …… 153
18.4 小结 …… 155
第 19 章 更多有关字符串的内容 …… 157
19.1 字符测试函数 …… 158
19.2 大小写测试函数 …… 158
19.3 大小写转换函数 …… 159
19.4 字符串函数 …… 159
19.5 小结 …… 162
第 20 章 更高级的数学函数 …… 165
20.1 简单的数学函数 …… 166
20.2 更多的转化 …… 167
20.3 三角函数和对数函数 …… 168
20.4 获取随机数 …… 169
20.5 小结 …… 171

第四部分 C 程序和大量数据

第 21 章 C 语言如何处理列表 …… 175
21.1 复习数组 …… 176
21.2 数组赋值 …… 179
21.3 有关数组的更多内容 …… 180
21.4 小结 …… 180
第 22 章 搜索数据 …… 183
22.1 填充数组 …… 184
22.2 搜索 …… 185
22.3 小结 …… 188
第 23 章 排序 …… 191
23.1 排序 …… 192
23.2 加快搜索 …… 196

23.3　小结 …… 199
第 24 章　我的名字叫指针 …… 203
24.1　内存地址 …… 204
24.2　定义指针变量 …… 205
24.3　使用取值运算符* …… 207
24.4　小结 …… 208
第 25 章　数组和指针有什么不同 …… 211
25.1　数组名是指针 …… 212
25.2　在数组中取值 …… 213
25.3　字符和指针 …… 214
25.4　小心字符串的长度 …… 214
25.5　指针数组 …… 216
25.6　小结 …… 217
第 26 章　有效管理内存空间 …… 221
26.1　使用堆 …… 222
26.2　为什么需要堆 …… 223
26.3　分配堆 …… 224
26.4　堆内存分配失败 …… 227
26.5　释放堆内存 …… 227
26.6　多次分配 …… 228
26.7　小结 …… 230
第 27 章　结构体 …… 233
27.1　定义结构体 …… 234
27.2　在结构体变量中存放数据 …… 238
27.3　小结 …… 240
第五部分　用函数组织程序
第 28 章　把数据存到硬盘上 …… 245
28.1　硬盘文件 …… 246
28.2　打开顺序文件 …… 247
28.3　访问顺序文件 …… 248
28.4　小结 …… 251
第 29 章　另一种保存文件的方式 …… 255
29.1　打开随机文件 …… 256
29.2　访问随机文件 …… 257
29.3　小结 …… 260
第 30 章　用函数来组织程序 …… 263
30.1　用C函数来组织程序 …… 264
30.2　局部变量和全局变量 …… 267
30.3　小结 …… 269
第 31 章　在函数间共享数据 …… 273
31.1　传递参数 …… 274
31.2　传递实参的方法 …… 274
31.2.1　按值传递 …… 275
31.2.2　按地址传递 …… 276
31.3　小结 …… 280
第 32 章　让函数更完美 …… 283
32.1　返回值 …… 284
32.2　返回的数据类型 …… 286
32.3　声明函数原型 …… 287
32.4　结语 …… 289
32.5　小结 …… 289
附录 A　你可以飞得更高 …… 291
附录 B　玩 C 语言写的 21 点游戏 …… 293
附录 C　ASCII 表 …… 301

Part 1 第一部分

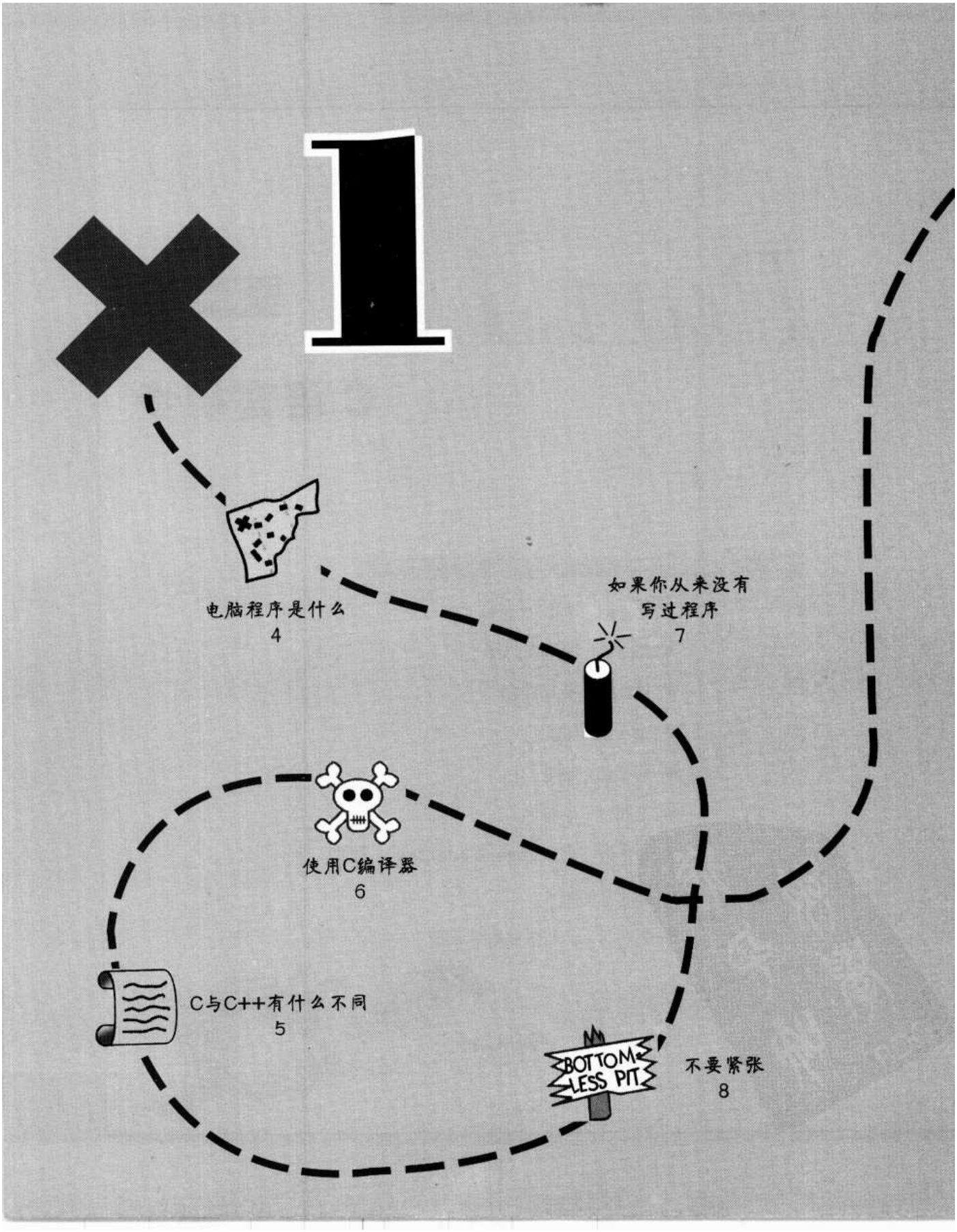
C 语言初步

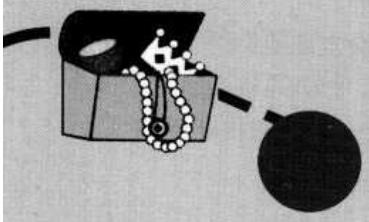
本部分内容

- 第 1 章　初识 C 语言
- 第 2 章　从何处入手
- 第 3 章　代码注释
- 第 4 章　输出
- 第 5 章　变量
- 第 6 章　字符串
- 第 7 章　#include 和 #define
- 第 8 章　输入
- 第 9 章　C 怎么做数学运算



1


电脑程序是什么
4
如果你从来没有
写过程序
7
使用C编译器
6
C与C++有什么不同
5
BOTTOM
LESS PIT
不要紧张
8



第1章

初识C语言

值得一学，充满乐趣

本章内容

什么是程序
4

编写C程序需要什么
5

编程过程
6

使用C语言
7

趣闻轶事
贝尔实验室开发了C语言。

虽然总是有些人认为学习和使用C语言很困难，但你很快就会发现他们错了。C语言被认为是一种神秘的编程语言，而它确实也可以变成这样。但是风格良好的C程序与用其他语言写出的程序一样易于理解。现在社会对C程序员的需求很旺盛，而且这种需求似乎永不衰竭。

如果你从未写过程序，本章会从最基本的知识开始，教你入门的编程概念，解释什么是程序，并简要介绍C语言的历史。你很兴奋吧？C语言可是个功能强大的编程语言。

1.1　什么是程序

电脑并不聪明。在你智商最低的时候，你的智力也超出电脑几光年。电脑的唯一优势是它遵守你的指令。电脑可以连续几天地处理你提供的数据，既不会感到厌烦也不会要加班费。

电脑不能自己决定做什么。电脑不能独立思考，所以程序员（programmer，告诉电脑要做什么的人）必须给电脑极其详细的指令。不发指令，电脑就毫无用处。若没有详细的指令，电脑无法处理你的工资表，正如汽车不能自己发动并在街上行驶一样。为了让电脑执行某项具体任务而提供给它的详细指令集合就是程序（program）。



说明　字处理程序、电脑工资表系统、电脑游戏和电子数据表都只不过是电脑程序。没有这些程序，电脑只能呆在那里，不知所措。比如，字处理程序就包含了一系列详细的指令，它们用计算机语言（如C语言）写成，能准确地告诉电脑怎样处理文字。编程就是让电脑执行程序中提供的指令。

你可以为电脑购买成千上万个程序，但是若公司想让电脑执行某项特定的任务，就要雇用程序员来编写符合要求的程序。你可以让电脑做很多事情，但却有可能找不到一个完全满足需求的程序。本书将把你从这种困境中拯救出来。学习了C语言之后，你就可以编写程序了，程序中包含的就是告诉电脑如何运转的指令。



线索　电脑程序告诉电脑如何执行你想要的操作。就像厨师需要食谱才能做成一道菜一样，程序需要指令来产生结果（见图1-1）。食谱就如同一系列详细的指令，写得好的话，就能清楚描述完成一道菜所需步骤的正确顺序和做法。程序之于电脑也正是如此。

当运行（run）或执行（execute）程序时，会产生输出（output）。做好的菜是食谱的输出，而工资表或字处理程序是运行程序产生的输出。

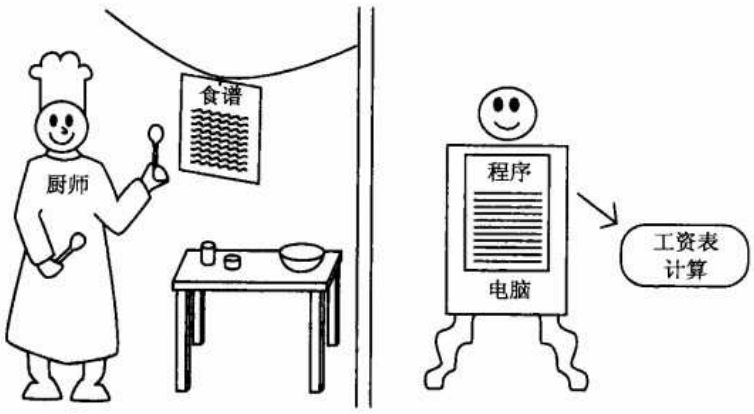


图1-1 正如厨师需要食谱才能烹饪，电脑要有程序才知道接下来要做什么

1.2 编写C程序需要什么

安装了C编译器（C compiler）之后，才能够编写并执行C程序。C编译器获取你编写的C程序并编译（compile，是一个技术术语，指把程序转换为电脑可识别的内容），它会在你准备查看结果的时候运行编译后的程序。如今的C编译器比几年前的高级许多。它们提供了全屏编辑、下拉菜单以及在线帮助，为初级程序员提供了许多的帮助。

现在几乎找不到单独的C编译器了。大多数情况下，C编译器捆绑（bundled，也是个术语，即包括）在C的一个更高版本C++中。因此，当你去买C编译器时，几乎总会发现C和C++编译器组合在一起。买下组合的C++/C软件包，你就拥有了C编译器，甚至为日后学习C++也做好了准备。



说明 本书最后一章简要描述了C++与C的不同之处，以及它对C的改进。

如今最流行的C编译器是*Turbo C++* 和*Borland C++*，二者都是Borland公司[①]开

① Borland公司是一家总部位于美国加利福尼亚州的软件公司，以提供软件开发生命周期所需的各种工具。2009年5月6日，Micro Focus宣布以7500万美元收购Borland。另外，Turbo C++是Borland C++的前身。

发的。*Borland C++*在Turbo C++的基础上增加了许多额外的程序，供高级的C和C++程序员使用。

微软公司提供了强大的C++和C编译器*Visual C++*。像Borland C++一样，Visual C++提供了高级的DOS、Windows和Windows NT编程。也可以用Turbo C++编写Windows程序，但Turbo C++没有像Borland C++和Visual C++那样附加很多的Windows编程工具。

顺便提一下，编写Window程序非常难，尤其对于编程初学者来说。别想一口吃个胖子。首先学会编写简单的基于DOS的C程序（本书就够用了），然后可以逐渐迁移到更难的C++和Windows编程。

市场上还有其他的C编译器厂商，但是Borland公司和微软公司的C编译器在C程序员中最受欢迎。



警告　你编写的C程序叫做源代码（source code）。编译器获取C源代码并将其翻译为机器语言（machine language）。电脑是由成千上万个开（on）或关（off）的电路开关组成的。因此，电脑最终接收的指令必须是二进制（binary）形式。前缀*bi*的意思是二（two），而电路的两种状态被称为二进制状态（binary state）。用C编译器来把你的C程序转换为1和0（表示内部的开或关）自然要比你自己亲手愚公移山要容易得多。

1.3　编程过程

大部分人写程序时按照下面的基本步骤进行。

(1) 确定程序要做什么。

(2) 使用一种编辑器（editor）编写并保存编程语言指令。编辑器类似于字处理器（尽管没有那么花哨），可以创建和编辑文本。所有流行的C编译器都包含一个集成在内的编辑器和编程语言编译器。所有的C程序文件名都以.C扩展名结束。

(3) 编译程序。

(4) 检查程序错误。如果存在错误，改正它们并返回步骤3。

(5) 执行程序。



说明　电脑程序中的错误称为bug。修正错误称为调试（debugging）程序。

如今的C编译器，如Turbo C++，使你可以很容易地执行这5个步骤，所有步骤都在同一环境下完成。例如，如果安装了Turbo C++，就可以使用Turbo C++的编辑器，编译程序、查看并修改错误、运行程序以及查看结果，所有这些都发生在同一屏幕下，使用统一的菜单。



请跳过这里，这有点儿难 如果你从未编过程序，那么上述这些可能听不太明白。不必紧张。现在的大多数C编译器带有方便的教程，告诉你怎样使用编译器的编辑器和编译命令。

简而言之，编程时大多数C编译器只需要你做这些：启动编译器，键入程序，然后选择菜单中的**Run**。Turbo C++有快捷键Alt+R（按住Alt键并按下R键，然后一起松开），之后回车编译并运行程序。编译器负责程序的编译和执行，遇到错误时它会通知你。



线索 许多时候，C编译器可以找出程序中的bug。例如，如果你拼错了一条命令，C编译器会在编译程序时通知你。

趣闻轶事
1983年，ANSI（美国国家标准学会）成立了X3J11委员会，制定出C语言的一个标准版本。它就是ANSI C。

万一现在你还没有完全明白编译器的作用，我打个比方。可以将源代码看成电脑需要的原始材料，编译器就像一台机器，负责把原材料转化成最终产品，即电脑可理解的编译后的程序。

1.4 使用C语言

C语言是现今最流行的一种编程语言。由于C语言的版本多种多样，*ANSI*委员会为所有版本开发了一套规则（*ANSI C*）。只要你使用ANSI C编译器运行程序，那么可以确保能在几乎所有装有ANSI C编译器的电脑上编译此C程序。只要设置恰当，你可以使大多数的编译器（包括Borland公司的和微软公司的）与ANSI C兼容。（请参阅编译器手册。）



线索 只要你编译了C程序，就可以在任何一台与你的电脑兼容的电脑上运行编译后的程序，而不管此电脑上是否有ANSI C编译器。

C语言比大多数编程语言更高效。它也是一种相对较小的编程语言。换句话说，你不必学习太多的C语言命令（command）。通过学习本书你将了解到C语言的命令和其他元素，如运算符、函数和预处理器指令等。



警告　仔细考虑一下并把思想状态转移到C语言上，因为下一章将开始你的第一个C程序之旅。



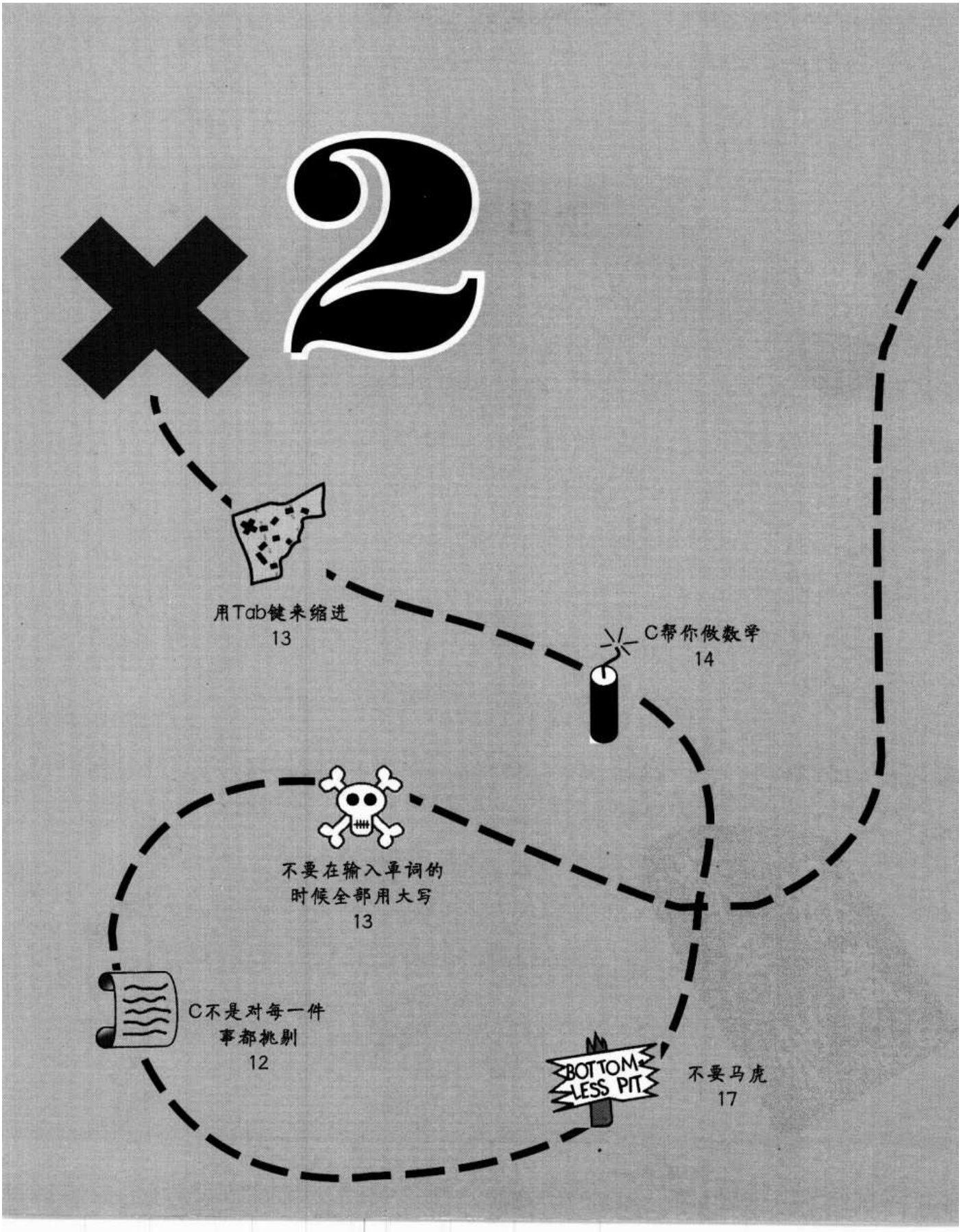
奖励

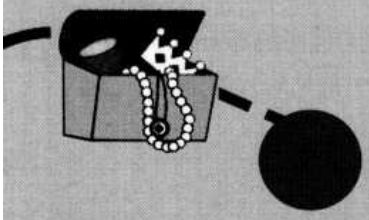
- 获取C编译器并在你的电脑上安装它。大多数编译器带有快速教程，能帮助你将编译器装载到电脑硬盘上。
- 学习C编程语言。这正是本书的目的！当你逐步熟悉C语言知识后，注意要用ANSI C命令，不要用某些编译器专用的C函数，因为这些函数在其他编译器上可能不存在。



陷阱

不要紧张，因为C语言很容易编写，并且通常C编译器有很多特性你不必知道。


×2
用Tab键来缩进
13
C帮你做数学
14
不要在输入单词的
时候全部用大写
13
C不是对每一件
事都挑剔
12
BOTTOM
LESS PIT
不要马虎
17



第2章

从何处入手

从main()函数开始

本章内容

概述
12

main()函数
13

数据类型
14

本章中，你将看到第一个C程序！不要试图理解这里讨论的C程序的每一个字符。放松些，先熟悉一下C的外观。马上你就要开始认识C程序中常见的元素了。

2.1　概述

本节向你展示一个简短但完整的C程序，并讨论附录B中出现的另一个程序。两个程序中包含了相同元素，也有不同的元素。第一个程序相当简单，如下所示：

```
/* Prints a message on the screen */
#include <stdio.h>
main()
{
  printf("This C stuff is easy!\n");
  return 0;
}
```

如果你在C编译器的编辑器中键入这段程序，编译程序并运行它，将会在屏幕上看到这条消息：

```
This C stuff is easy!
```



说明　看上去要做很多工作才能输出这条消息！实际上，在这7行代码中，只有以printf开头的那一行产生了输出，其他行提供的只是大部分C程序共有的例行内容。



线索　真正的长程序可以看附录B。尽管附录B中21点游戏用了好几页纸，但是它也包含了你刚看到的这个小程序里的共同的元素。

浏览刚才讨论的两个程序，找找它们的相似点。你注意到的第一个相似处可能是大括号{}、圆括号()和反斜杠\的使用。向C编译器键入程序时要小心。C语言很挑剔，它不能容忍在需要圆括号时键入方括号[。



说明　C语言并非对每样东西都很挑剔。例如，在C程序中的大多数空格是为了让人更易读，而不是让编译器更易读。在程序中加上空行和代码缩进可以帮助改善程序的可读性，使你更容易查找。



线索 要使用Tab键来缩进而不要用一连串的空格。大多数C编辑器允许调整制表符间距（按下Tab键时出现的空格数）。一些C程序的行很长，所以设置Tab为3就能提供足够大的缩进且不会使一行太长。

C要求所有命令和预定义函数使用小写字母（2.2节将学习什么是函数）。只有在#define行和打印的消息中才使用大写字母。

2.2 main()函数

C程序中最重要的部分是main()函数。前面讨论的两个程序都有main()函数。main()是一个C函数（function）而不是C命令，虽然现在这个区分不是很重要，但你还是应该知道这一点。函数是指C自带的或者由你编写的执行某些任务的例行程序。C程序由至少一个函数构成。每个程序必须总是包含一个main()函数。函数名后面带有括号，可以由此与命令相区别。下面这些是函数：

```
main          calcIt()      printf()      strlen()
```

而下面这些是命令：

```
return        while         int     if        float
```

在其他C编程的书或手册中，作者可能会省略函数名后面的括号。例如，你可能会看到printf函数而不是printf()。你很快就会识别函数名，所以这种区别不会给你带来太大影响。大多数时候，作者都会想尽可能地区分函数和非函数，所以你看到的更多是带括号的函数。



警告 刚才列出的一个函数calcIt()含有一个大写字母。然而，2.1节说过在C语言中，所有命令和预定义函数应该使用小写字母。如果一个名称有多个部分，如doReportPrint()，通常把每个分隔词以大写字母开头，这样可以提高可读性（函数名中不允许出现空格）。不能把单词全部用大写，为了可读性偶尔用一个大写字母还是可以的。



线索 必需的main()函数和所有C语言提供的函数名必须使用小写字母。自己写的函数可以用大写字母，但是大多数C程序员遵循使用小写字母函数名的惯例。

正如我们总是从一本书的第1章开始阅读，电脑也总是从main()开始运行程序。就算main()在程序中不是排在第一个函数的位置，main()仍然决定着程序执行的开始。因此，在你编写的每个程序中都要让main()函数排在第一个。后面几章中的程序都只有一个main()函数。等你具备了一些C编程知识后，就会知道还可以在main()函数后面添加其他函数，提高程序的运行能力。

在单词main()后面，总会看见左大括号{。当你看到一个对应的右大括号}时，main()就结束了。main()函数内部也可能会有其他的大括号对。作为练习，可以再看一下附录B中的长程序。main()是第一个带有代码的函数，还有其他几个函数跟在后面，每个都有大括号和代码。



说明　几乎每个C程序都需要语句#include <stdio.h>。它用来打印和获取数据。目前，总是把这条语句放在main()前面。在第7章中，你将明白为什么#include很重要。

2.3　数据类型

C程序中的数据必须由数字、单词和字符构成。程序将数据处理成有意义的信息。虽然说有很多不同类型的数据，但是以下3种数据类型是目前在C编程中最常用的：

- 字符
- 整数
- 浮点数（也称实数）



请跳过这里，这有点儿难　你也许会埋怨："我得学习多少数学知识？我觉得不太划算！"不用担心，因为C语言为你解决了数学问题。你不必先学会2加2才写C程序。但是，你必须理解数据类型，这样才能在程序中正确使用它们。

2.3.1　字符

趣闻轶事
开发了ANSI C的美国国家标准学会（ANSI）开发了ASCII表。

C语言中的字符（character）是电脑能表示的任意一个字符。电脑能识别256个不同的字符，每个字符都能在*ASCII*表中找到，请参阅附录C。电脑能表示的任何东

西都能作为字符。下面列出的全部都能看成是字符。

```
A   a   4   %   Q   !   +   =   ]
```



线索 空格键也会产生一个字符。就像C语言需要记录下字母表的字母、数字和所有其他字符一样，它也必须记录下程序所需的任何空格。

可以看到，每个字母、数字和空格对C语言来说都是字符。当然，4似乎是一个数字，是的，有时候它是数字，但有时候也可以当作字符。如果你指明某个4是字符，那么就不能用它进行数学计算；如果你指明另一个4是数字，就可以用这个4进行数学运算。对特殊符号来说也是一样的。加号（+）是一个字符，但它也可以执行加法运算。不好意思，说着说着又谈到数学方面了。

C语言的所有字符数据都括在撇号'中。一些人把撇号称为单引号。撇号把字符数据与其他类型的数据区分开来，如数字和数学符号。例如，在C程序中，下面所列的都是字符数据：

```
'A'   'a'   '4'   '%'   ' '   '-'
```

下面所列的都不是字符数据，因为它们都没有带撇号：

```
A   a   4   %   -
```



线索 下面所列的都不是有效字符。只有单个字符可以放在撇号中，多个字符不可以。

```
'C is fun'   'C is hard'   'I should be sailing!'
```

本章的第一个程序包含字符'\n'。一开始，你可能认为\n不是一个字符，实际上，它是少数的几个两字符组合，C语言将这些组合解释成单个字符。稍后会详细介绍这一点。

如果你需要指定多个字符（除了将要学习的特殊字符外，还有比如刚才描述的\n），则需要把字符放在引号（""）中。多个字符的组合称为字符串（string）。下面是一个C语言字符串：

```
"C is fun to learn."
```



说明　关于字符和字符串知道这些就够了。本书稍后将介绍如何在程序中使用它们。当你读到如何在变量中存储字符时，就会明白撇号和引号为何如此重要了。

2.3.2　数字

你以前可能没想到过，数字有许多不同的大小和形状。不论数字什么样子，C程序必须有一种存储数字的方式。你必须把数字存储在数值变量中。在查看变量之前，检查一下数字的类型会对你有所帮助。

整数是指没有小数的完整的数。整数没有小数点。（记住这条规则：就像国会多数议员说话没有要点一样，整数没有小数点。）任何没有小数点的数都是整数。下面所列的都是整数：

```
10      54      0      -121      -68      752
```



警告　不要让整数以0开头（除非数字是0），否则C语言会认为你输入的数字是16进制的或8进制的。16进制和8进制有时也称为以16为基和以8为基，它们都是表示数字的奇特的方式。053是一个8进制数，0x45是一个16进制数。如果你不明白这些，现在只需要记住如果在整数前加上0，C语言就会打开16/8进制的开关。

带小数点的数字称为浮点数（float point number）。下面所列的都是浮点数：

```
547.43      0.0      0.44384      9.1923      -168.470      .22
```



线索　如你所见，浮点数允许以0开头。

选择使用整数还是浮点数取决于程序要处理的数据。某些值（如年龄和数量）可以用整数，而某些值（如钱的数量）则需要用浮点数。在内部，C语言存储整数与存储浮点数的方式不同。如图2-1所示，浮点数占用的内存通常是整数的两倍。[①] 因此，如果可能请尽量使用整数，把浮点数留给那些需要小数点的值吧。

① 在16位的机器上通常是这样，但是在32位的机器上，二者用的内存一般都是4个字节。——译者注

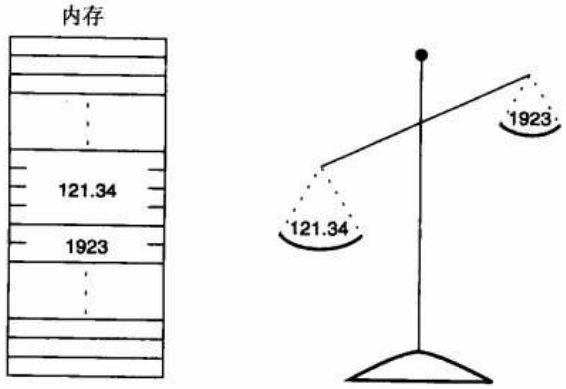


图2-1 浮点数通常比整数占用的内存多



说明 图2-1展示了无论存储的值是大还是小，整数通常比浮点数占用的内存少[①]。不管什么时候，大的邮政信箱都有可能会比小信箱收到的邮件少。信箱的容量并不会影响信箱存放的东西。C语言的数字存储内存的大小不会受到数值的影响，但会受到数字类型的影响。

不同的C编译器为整数和浮点数分配不同大小的内存。随后你将会看到，有几种办法可以查明C编译器为每种类型的数据到底分配了多少内存。



奖励

- 让大小写锁定键处于关闭状态！大多数C命令和函数要求使用小写字母。
- 在C程序中加入空格使其更具可读性。
- C函数在名字后面必须有括号。C程序由一个或多个函数构成。总需要有一个`main()`函数。C语言在执行其他函数之前要执行`main()`。
- 如果使用字符，要用单引号引起来。字符串要在引号中。整数不带小数点。浮点数带有小数点。



陷阱

- 输入代码的时候一定不要马虎。当C语言需要某个特定的字符比如单括号时，就一定不能错写为方括号。
- 不要让整数以0开头，除非整数是0本身。

① 在16位的机器上。——译者注

2.4　小结

本章主要使你熟悉C程序的大体样式，主要是熟悉包含了可执行的C语句的main()函数。如你所见，C语言格式比较自由，对空格的使用并没有过分讲究。但是，C语言很讲究用小写字母，它要求所有的命令和函数用小写字母拼写，如printf()。

现在，请不必担心在本章中看到的代码的细枝末节。本书的后续部分会解释所有这些细节。

1. 代码范例

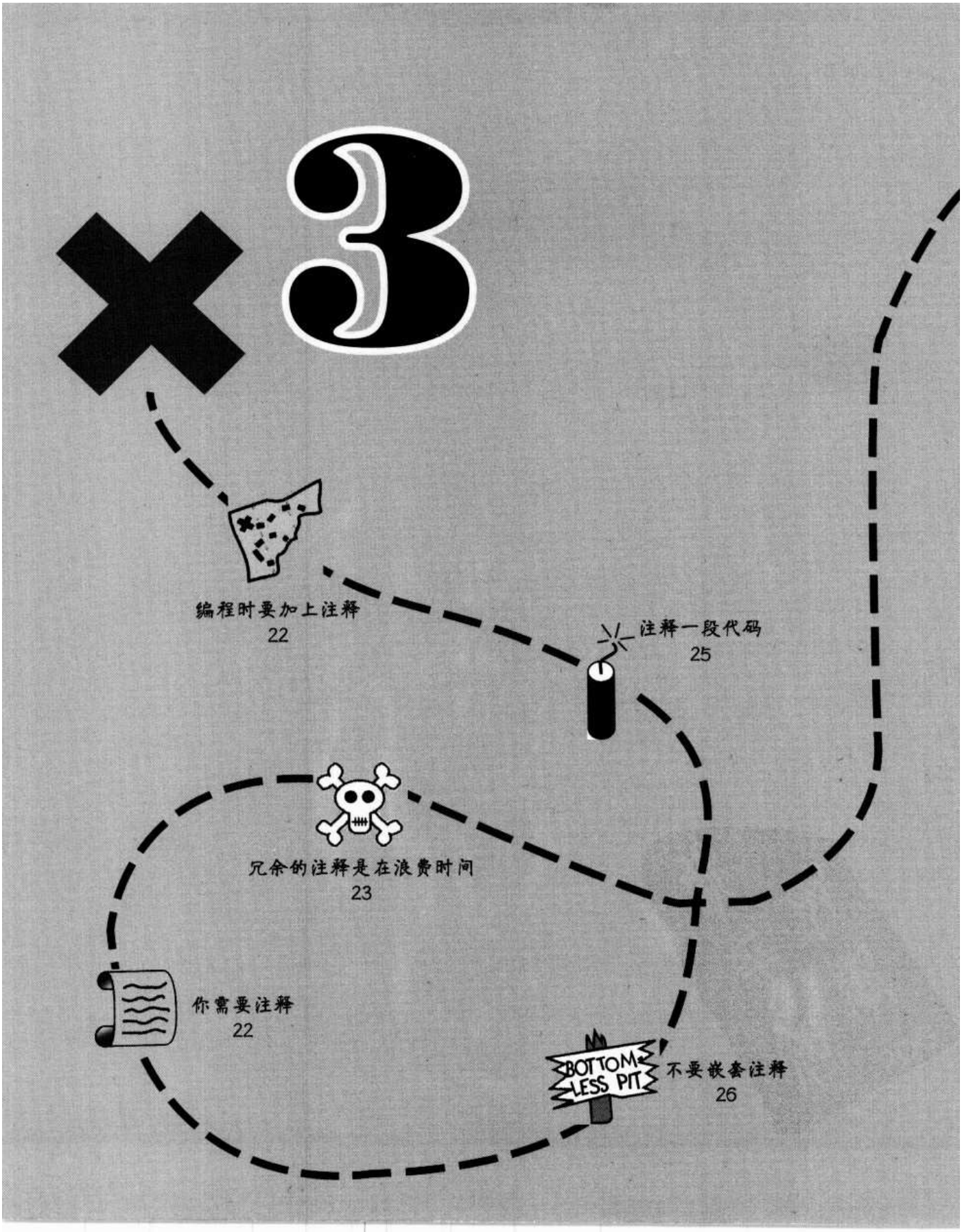
```
/* Prints a character and some numbers */
#include <stdio.h>
main()
{
  printf("A letter grade of %c\n", 'B');
  printf("A test score of %d\n", 87);
  printf("A class average of %.1f\n", 85.9);
  return 0;
}
```

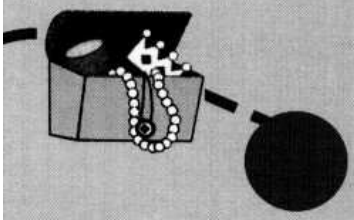
2. 代码分析

这个短程序只是在屏幕上打印了3行消息。每行消息中包含了本章提到的3种数据类型中的一种，即字符（B）、整数（87）和浮点数（85.9）。

程序中唯一由程序员写的函数是main()函数。左右大括号总是把main()的代码括起来，而任何你添加到程序中的其他函数也都是这样。你将会看到另一个函数printf()，它是内置的C函数，用于产生输出。下面是程序的输出：

```
A letter grade of B
A test score of 87
A class average of 85.9
```


3
编程时要加上注释
22
注释一段代码
25
冗余的注释是在浪费时间
23
你需要注释
22
BOTTOM LESS PIT
不要嵌套注释
26



第3章

代码注释

注　　释

本章内容

给代码加注释
22

注释详述
23

留白艺术
25

另一种注释方式
26

电脑必须能够理解你的程序。因为电脑是不会说话的机器，所以必须小心地拼写完全正确的C命令，并按照你想要的执行顺序输入。然而，其他人也会读你的程序。另外，你会经常修改程序，并且如果你是为一家公司写程序，公司的需求也会经常改变。因此，必须确保你的程序能被他人和电脑理解。因此，你应该为程序写文档，解释它们做了什么。

3.1　给代码加注释

趣闻轶事
注释有时叫做说明或文档。

整个C程序从头至尾都应加上注释（comment）。注释是散布于程序中的解释性信息。如果你写程序来计算工资表，那么程序的注释就会解释薪水总额、国税、地税、社保费等所有数字是怎么计算出来的。



说明　如果你写的程序只是你自己使用，就不需要注释了，对吗？不是的。C语言是一种晦涩的编程语言，即使程序是你自己写的，但是过后你很可能也会看不懂它。



线索　在写程序的同时要加上注释。现在就要养成这个习惯，因为程序员很少在程序写好后再回头添加注释。等到必须修改程序时，就会后悔没有早给程序加注释。

在写程序的同时加注释还有另一个好处。写程序时，经常需要查看前面写的语句。你不必重新去理解所写过的C代码，可以通过浏览注释很快地找到要看的代码部分。如果没有加注释，那么就不得不译解每一段C代码。

程序维护（maintenance）是随时间流逝不断修改程序的过程，它会修正隐藏的错误，并调整程序以适应环境的改变。如果你为某家公司写了一个工资表程序，这家公司可能某一天会改变计算薪水的方式，于是你（或别的程序员）就得修改工资表程序，与公司的新制度保持一致。注释可以方便程序的维护。有了注释，就可以快速浏览程序并找出需要修改的地方。

注释不是C的命令。C语言会忽略程序中的所有注释。注释是给人看的，而注释之间的程序语句是给电脑看的（参见图3-1）。

考虑下面的C语句：

```
return ((s1 < s2) ? s1 : s2);
```

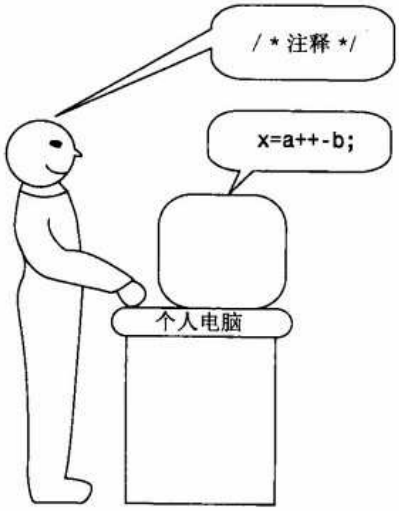


图3-1 注释给人看，而C程序语句给电脑看

你还没掌握C语言，然而，即使你已经掌握了，这条语句还是要费些脑筋才能理解。下面这样写不是更好吗？

```
return ((s1 < s2) ? s1 : s2); /* Gets the smaller of 2 values */
```

3.2节将解释注释的语法，但现在，你明白在/*和*/之间的信息是注释。

注释要口语化且有别于C代码。不要只是为了注释而写注释。下面语句的注释是毫无用处的：

```
printf("Payroll");  /* Prints the word "Payroll" */
```



警告 你还没掌握C语言，不过你仍然不需要上面的注释行！冗余的注释是在浪费时间，不会给程序带来任何好处。要为可能需要读你的程序的人（包括你自己）加注释，解释程序在做什么。

3.2 注释详述

C语言的注释以/*开头并以*/结束。注释可以在程序中跨越好几行，并且可以放置在程序的任何地方。下面的每一行都包含了注释：

```
/* This is a comment that happens to span two lines
before coming to an end */

/* This is a single-line comment */

for (i = 0; i < 25; i++)  /* Counts from 0 to 24 */
```



说明　注释可以在编程语句之前或之后独立成行。位置的选择取决于注释的长度和注释所描述的代码量。

附录B中的21点游戏程序包含了所有类型的注释。通过通读程序中的注释，你不需要看C代码就能知道程序做了什么。

不要对每一行加注释，通常每隔几行才需要注释。许多程序员喜欢在一段代码前放一段多行的注释，然后在需要的行中插入短一些的注释。下面是一个带有各种注释的完整程序：

```
/* Written by: Perilous Perry, finished on April 9, 1492 */
/* Filename: AVG.C */
/* Computes the average of three class grades */
#include <stdio.h>
main()
{
  float gr1, gr2, gr3;  /* Variables to hold grades */
  float avg;            /* Variable to hold average */
  /* Asks for each student's grade */
  printf("What grade did the first student get? ");
  scanf(" %f", &gr1);
  printf("What grade did the second student get? ");
  scanf(" %f", &gr2);
  printf("What grade did the third student get? ");
  scanf(" %f", &gr3);

  avg = (gr1 + gr2 + gr3) / 3.0;  /* Computes average */
  printf("\nThe student average is %.2f", avg);
  return 0;  /* Goes back to DOS */
}
```

许多公司要求程序员在其所写程序顶部的注释中加入自己的名字。如果将来需要修改程序，可以找最初的程序员来帮忙。在程序的开始处写上它在硬盘上保存的文件名也是一个好习惯，这样当你拿到打印出来的程序时可以在硬盘上找到它。



说明　这本书在有些地方可能注释得过多了，尤其是在开始的几章里。你还不熟悉C语言，每个细小的解释都有很大的帮助。



请跳过这里，这有点儿难 测试代码时，你可能会发现用/*和*/注释掉一段代码很有用。这样做能让C语言忽略这段代码，于是就可以全神贯注于正在处理的代码。但是，不要注释掉一段已经含有注释的代码，因为注释不能嵌套。C语言遇到第一个*/就认为是注释已结束，而当C语言发现下一个*/前面没有表示开始的/*时，就会报错。

3.3 留白艺术

空白（white space）指很多程序中存在的空格和空行。在某种程度上，空白在提高程序的可读性方面比注释更重要。人们在看C程序时需要有空白，程序代码不能都挤在一起。看一看下面的程序：

```
#include <stdio.h>
main(){float s,t;printf("How much do you make? ");scanf(" %f",
&s);t=.33*s;printf("You owe %.2f in taxes.",t);return 0;}
```

对编译器来说，这是一个相当好的C程序，但对正在看程序的人来说就不是了，尽管代码很简单并且不难理解。下面的程序即使没有注释，也要容易理解得多：

```
#include <stdio.h>
main()
  {
    float s, t;

    printf("How much do you make? ");
    scanf(" %f", &s);

    t = .33 * s;
    printf("You owe %.2f in taxes.", t);
    return 0;
}
```

这个代码清单与前一个是一样的，只是它包含了空白，而前一个没有。程序的实际长度不能决定其可读性，空白的数量对程序的可读性来说才是决定性的（当然，加上一些注释也能改善程序，但是这个练习是想说明有没有空白会有多么大的区别）。



说明　你可能想知道为什么在挤在一起的那个程序的第一行，即带有#include的行，在右尖括号之后没有写代码。毕竟，如果在它后面再加上代码，程序的可读性就更差。我差点儿想那么干！许多C编译器不允许在#include（或者任何其他以#开头的语句）后面加上代码。某些C编译器甚至不允许在这样的行尾加注释，尽管现在的C编译器一般都允许在那里加注释了。

3.4　另一种注释方式

现在许多C编译器支持一种原本为C++程序设计的注释，即以两个斜杠（//）开头且只在行尾结束。

下面是新的注释方式：

```
// Short program!
#include <stdio.h>
main()
{
  printf("Looking good!");  // A message
  return 0;
}
```

不过，本书采用的是/*和*/形式的注释方式。



奖励

- 编程的3条规则是注释、注释、再注释，尽可能多地使用注释。
- 使用注释时，以/*开始，以*/结束。
- 新的注释方式是以//开始注释行的。但是，这种注释还没有被ANSI C认可。



陷阱

- 不要使用冗余的注释。没有价值的注释，只会浪费你宝贵的编程时间。
- 不要嵌套注释。如果你要注释掉一段程序，必须确保这段代码不含有注释。
- 不要写几乎没有空白的程序。需要时尽可能多地在程序中使用缩进和空行，对代码行进行分组。当你学习了C语言的更多知识之后，你将学会在哪里加空白能提高程序的可读性。

3.5 小结

你必须为程序加注释，它们不是给电脑看的，是给人看的。尽管C程序有点晦涩，但注释可以为你消除许多混乱。注释用来说明C代码正在做什么。在/*和*/之间的任何东西都是C语言的注释。C语言编译执行时会忽略所有的注释，因为它知道注释是给人看的。

除了注释，在程序中加上许多空白能使程序更具可读性。如果程序都挤在一起而没有空行和有用的缩进，当你日后回来研究和修改代码时，就会觉得像在读一本不分段落的书。日后一旦要修改程序，注释可以使程序更易于维护，而足够多的空白能帮助你节省时间和精力。

1. 代码范例

下面是两行没有注释的代码：

```
scanf(" %d", &a);
yrs = (a >= 21) ? 0 : 21 - a;
```

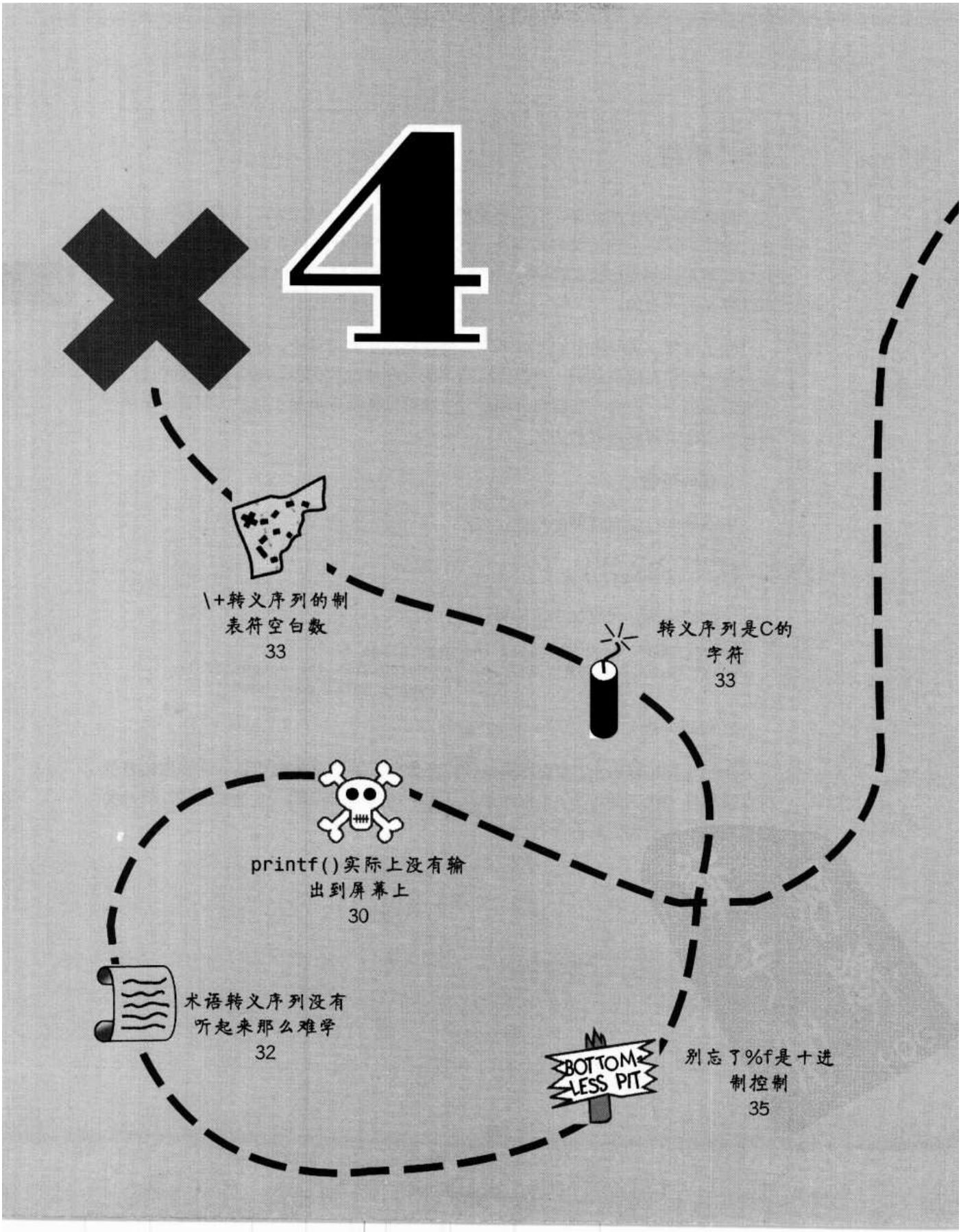
下面给这两行代码加上了注释：

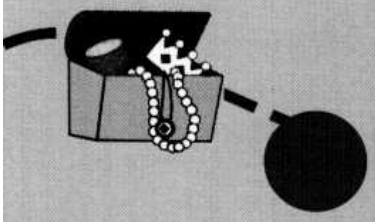
```
scanf(" %d", &a);  /* Gets the user's age */
yrs = (a >= 21) ? 0 : 21 - a;  /* Calculates the number of */
                               /* years until adulthood */
```

2. 代码分析

从这几行可以看出，C程序中的代码要做什么并不总是很明显。注释用简易的口头语解释了代码的行为。C程序中并非每一行都需要注释，但是多加些注释确实能更清楚地阐明代码的行为。

4


\+转义序列的制
表符空白数
33
转义序列是C的
字符
33
printf()实际上没有输
出到屏幕上
30
术语转义序列没有
听起来那么难学
32
BOTTOM
LESS PIT
别忘了%f是十进
制控制
35



第4章

输　出

使用printf()

本章内容

printf()做了什么
30

printf()的格式
30

打印字符串
31

转义序列
32

转化字符
33

如果任何人（包括你自己）都看不到程序的输出，那么程序就没有什么用了。所以最后，你必须要能够查看程序的输出结果。C语言用于输出的主要方法是使用printf()函数。没有其他的命令执行输出，但printf()函数是每个C编译器的一部分，并且是该语言中最常用到的特性。

4.1　printf()做了什么

简而言之，printf()在屏幕上产生输出。如图4-1所示，printf()把字符、数字和单词发送到屏幕上。printf()有很多选项，但是你不必精通所有这些选项（很少有程序员能做到这一点）才能使用printf()在屏幕上输出。

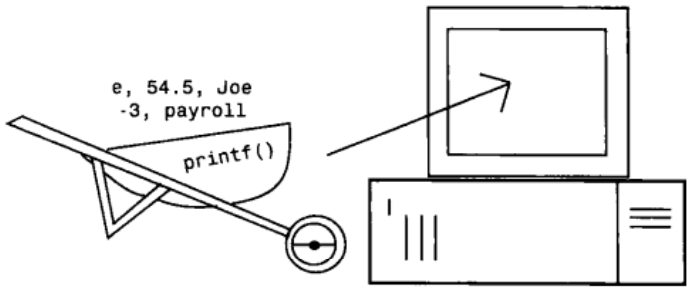


图4-1　printf()把字符、数字和单词发送到屏幕上

4.2　printf()的格式

printf()有许多种形式，但是一旦你习惯了它的格式，printf()就很容易使用。下面是printf()的一般格式：

```
printf(controlString [, data]);
```

本书常常在你第一次见到某命令和某函数时给出它们的格式。格式是指语句一般的形式。如果格式中有一部分出现在方括号中，如上面的data，那么这部分就是可选的。你不需要输入方括号。如果命令中需要方括号，则会在格式后面的文本中说明清楚。printf()需要一个controlString，但其后面的data是可选的。



警告 printf()实际上没有把输出发送到屏幕，而是发送到了电脑的标准输出设备（standard output device）。大多数操作系统，包括MS-DOS在内，都将标准输出发送到屏幕，除非你足够了解MS-DOS，能让它把输出发送到其他地方。大多数时候你可以忽略这个标准输出设备，因为你几乎总是想输出到屏幕上。后面要学习的其他C函数可以把输出发送到打印机和硬盘驱动器。



警告 你可能想知道为什么格式中的一些单词用斜体。这是由于它们是占位符（placeholder）。占位符是你提供的名称、符号或公式。在函数和命令的格式中占位符是斜体的，提醒你这个位置要用一些东西来替代。

趣闻轶事
C的设计者从FORTRAN语言中借鉴了格式控制字符串。

下面是一个printf()的例子：

```
printf("I am %d", 16);  /* Prints I am 16 */
```

因为C语言中的每个字符串必须出现在引号中（在第2章中提到），所以controlString必须在引号中。controlString后面的任何东西都是可选的，并且是根据想要打印的值取舍的。



说明 每个命令和函数后面都需要分号（;），C语言据此判断一行是否结束。括号和函数的第一行不需要分号，因为这些行中没有可执行的内容。所有printf()语句都要以分号结束。但是，不需要在main()后面加分号，因为你不会显式地让C语言去执行main()。但你需要显式地让C语言去执行printf()和其他许多函数。等你学了更多C语言知识，你将会更了解分号的放置。

4.3 打印字符串

字符串消息是用printf()打印的最简便的数据类型。你只需要把字符串放在引号中。下面的printf()语句在屏幕上打印一条消息：

```
printf("Read a lot");
```

电脑执行这条语句时，屏幕上会出现Read a lot。



说明　字符串Read a lot就是这个printf()中的constrolString。这里几乎没有什么控制（control），仅仅是输出。

下面两行printf()语句

```
printf("Read a lot");
printf("Keep learning");
```

可能不会产生你所期望的输出。实际产生的输出如下：

```
Read a lotKeep learning
```



线索　当printf()执行时，C语言不会自动把光标移到下一行。如果想让C语言在printf()之后转到下一行，你必须在controlString中插入一个转义序列。

4.4　转义序列

C语言包含许多转义序列，几乎在你写的每个程序中都要用到一些。表4-1列出了使用较多的转义序列。

表4-1　转义序列

代　码	描　述
\n	换行
\a	警报（电脑响铃）
\t	制表符
\\	反斜杠
\"	引号



说明　转义序列其实没有听起来那么难学。一个转义字符在C语言中存储为一个字符，并且它产生表4-1中描述的效果。例如，当C发送'\a'到屏幕时，电脑的铃声会响起，而不是真把字符\和a打印出来。

你将在printf()函数中看到很多转义序列。打印多行文本时，如果想要移到下一行，就必须键入\n。C程序会产生新行（newline），它会把闪烁的光标移到屏幕的下一行。在下面的两行printf()语句中，由于在第一个语句后面有\n，它们会分

两行打印消息。

```
printf("Read a lot\n");
printf("Keep learning");
```



请跳过这里，这有点儿难 \n可以放在第2行的开始处，输出结果相同。因为转义序列对C语言来说是字符，所以必须把它们放在引号中，这样C语言就知道转义序列是要被打印的字符串的一部分。下面的代码也会产生两行输出：

```
printf("Read a lot\nKeep learning");
```


因为引号标志字符串的结束，而反斜杠标志转义序列的开始，所以它们都有自己的转义序列。\a让电脑响铃，\t让输出后移一些空格。查看编译器手册，找出你的C语言所支持的其他转义序列。尽管表4-1中的转义序列几乎是最普遍的（且是与ANSI C兼容的），但不是所有的ANSI C编译器在所有的编译模式下都支持这些转义序列。例如，Visual C++在QuickWin模式下就不允许使用\a。

下面的printf()语句将输出注释里的内容：

```
printf("Ready\tSet\tGo!\n");       /* Ready     Set     Go! */
printf("Ring my charm!\a\n");       /* Ring my charm! <BEEP> */
printf("I said, \"No way.\"\n");    /* I said, "No way." */
printf("\\ means escape\n");        /* \ means escape */
```



线索 不同的编译器可能为\t转义字符产生不一样多的制表符空格。

4.5 转化字符

打印数字和字符时，必须准确告诉C程序如何打印它们。用转化字符（conversion character）来表明数字的格式。表4-2列举了C语言中最常用的转化字符。

表4-2 转化字符

转化字符	描 述
%d	整数
%f	浮点数
%c	字符
%s	字符串

当你想在字符串中打印一个值时，要在controlString中插入合适的转化字符，然后在controlString的右边，列出想要打印的值。图4-2是一个例子，它说明了printf()如何打印3个数字——一个整数、一个浮点数和另一个整数。

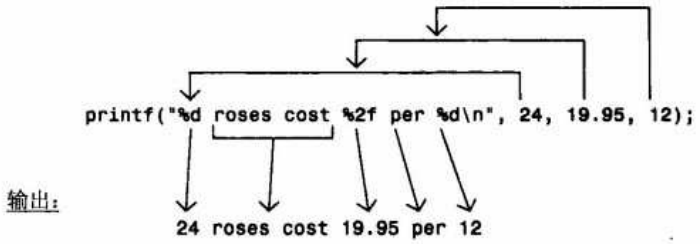


图4-2　printf()转化字符决定了怎样打印以及在哪里打印数字

字符串和字符也有自己的转化字符。尽管只打印字符串时可以不需要%s，但是当打印与其他数据组合的字符串时，就可能需要用%s。下面的printf()用各种转化字符打印几种不同类型的数据：

```
printf("%s %d %f %c\n", "Sam", 14, -8.76, 'X');
```

上述代码的输出如下：

```
Sam 14 -8.760000 X
```



说明　字符串Sam需要加上引号，所有的字符串都需要这样。而字符X需要加上单引号，所有的字符都是这样。



警告　当C语言遇到浮点数就会变得疯狂。尽管-8.76只有两个小数位，但C语言却坚持打印出6个小数位。

可以通过在浮点转化字符的%和f之间加上一个。来控制C程序如何打印浮点数值。图4-2使用了这种小数控制。尽管Printf()语句中给定的浮点数相同，不过运行输出的4个数却不相同：

```
printf("%f  %.3f  %.2f  %.1f", 4.5678, 4.5678, 4.5678, 4.5678);
```

C程序把浮点数四舍五入到在%.f转化字符中指定的小数位，并产生如下输出：

```
4.567800  4.568  4.57  4.6
```



线索 等你在下一章中学习变量时，转化字符会变得更有用。

最后一个要点：printf()的constrolString严格控制着输出结果。之所以在数字之间出现两个空格，是因为controlString中各个%f之间有两个空格。



奖励

- 如果想在屏幕上打印数据就用printf()。
- 每个printf()需要一个控制字符串，用来决定数据打印的结果。
- 打印字符串很容易。它们不需要特殊的格式代码或转化字符。
- 用转义序列实现换行、制表符、引号、反斜杠和电脑响铃。
- 用转化字符来控制数字打印的结果。



陷阱

- 不要指望C语言知道如何自动格式化数据。你必须自己使用转化字符。
- 不要忘记用%f的小数控制，除非你想让C程序把所有的浮点数都打印出6个小数位。

4.6 小结

printf()把数据输出到屏幕上。不能输出到屏幕的程序几乎没有用。你写的程序必须能够与键盘边的用户交流。

printf()需要一个controlString，它描述了其后面的数据格式。通过使用contrlString，能够准确指定数字和字符数据如何打印。你也可以打印转义序列，这是一种有时要用到的特殊的输出控制，比如要在输出的行尾换行。

1. 代码范例

看一看下面的部分代码清单：

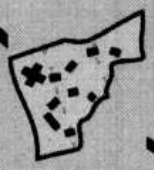
```
printf("%c %s %d %f %.2f\n", 'Q', "Hello!", 14, 64.21, 64.21);
printf("%c\n", 'Q');
printf("%s\n", "Hello!");
printf("%d\n", 14);
printf("%f\n", 64.21);
printf("%.2f", 64.21);
```

2. 代码分析

第一个printf()打印出5个数据值——一个字符、一个字符串、一个整数、两个浮点数。接下来的5行把这些值一次打印一个。最后一个浮点值的小数位数在最后一个printf()中指定，用来限制打印出来的小数位。下面是代码的输出：

```
Q Hello! 14 64.210000 64.21
Q
Hello!
14
64.210000
64.21
```

5



等号告诉C……
43



大多数C的变量在左大括号之后定义
42



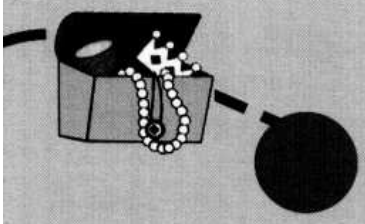
可以在变量中存储字符串
41



C程序中变量的大小不同
40



不要混淆数据类型和变量类型
44



第5章

变　量

使用变量

本章内容

变量类型
40

变量的命名
41

定义变量
41

在变量中存储数据
42

众所周知电脑可以处理数据。不管怎样，你必须有办法存储这些数据。与大多数编程语言一样，C语言把数据存储在变量（variable）中。变量仅仅是电脑内存中的盒子，用来保存数字或字符。第2章讲解了不同的数据类型：字符、字符串、整数和浮点数。本章讲解如何在程序中存储这些类型的数据。

5.1　变量类型

由于有几种不同类型的数据，所以C语言中也有几种不同类型的变量。任何一种变量都不能保存几种数据。只有整型变量才能保存整型数据，只有浮点型变量才能保存浮点数据，如此类推。



说明　在本章我们将电脑中的变量看成是邮局中的邮箱。邮箱的大小各不相同，且有彼此不同的编号。程序中变量的大小也不同，这取决于它保存哪种数据类型。此外，每个变量都有不同的名字与其他变量相区别。

第2章中所学的数据称为字面量数据（literal data）或常量数据（constant data）。特定的数字和字母不会改变。数字2和字符'x'始终是2和字符'x'。不过，要处理的大多数据都是要改变的。如年龄、工资和重量等数据都是变化的。如果想编写一个工资表程序，就需要一个方式来保存变化的数据。变量可以帮你实现。变量就是内存中用来存放数据的小盒子，里面的数值可以随时间改变。

有许多变量类型。表5-1列出了较常用的变量类型。注意，许多变量的数据类型（字符、整型和浮点型）与字面量数据很类似。毕竟，你必须有一个地方来存放整数，即使用整型变量。

表5-1　几种变量类型

名　称	描　述
char	存放字符数据，如'x'和'*'
int	存放整型数据，如1、32和-459。存放的数据在-32768和32767之间
long int	存放大于32767和小于-32768的整型数据
float	存放浮点数，如0.0003、-121.34和43323.4
double	存放极大和极小的浮点数。float只能存放从-3.4×10^{38}到3.4×10^{38}之间的数



警告 你可能注意到了，尽管有字符串字面量类型，但却没有字符串变量。C是少数几种没有字符串变量的一种编程语言，但是第6章会介绍如何在变量中存储字符串。

表5-1中的名称一列给出了编程时创建变量要用的关键字。也就是说，如果想要一个整型变量，就要用int关键字。最后你还需要知道如何给变量命名。

5.2 变量的命名

所有的变量都有名字，而且由于是你负责对它们命名，你必须掌握命名规则。所有的变量名必须不同。在同一个程序中不能有两个相同名字的变量。

趣闻轶事
第一个编程语言没有变量名。只能通过数值的内存位置来引用变量。

变量名可以包含1到32个字符。变量名必须以字母开头，后面可以是字母、数字或下划线的组合。下面所列的都是合法的变量名：

```
myData   pay94   age_limit   amount   QtlyIncome
```



线索 C语言允许变量名以下划线开头，但是你不应该这么做。因为C语言的一些内置变量以下划线开头，如果你的变量也以下划线开头，就有可能与某个内置变量重名。

下面这些变量名是不合法的：

```
94Pay     my Age     rate*pay
```

你应当明白为什么这些变量名不合法。第1个94pay，以数字开头；第2个变量名my Age，含有一个空格；而第3个变量名rate*pay，含有一个特殊字符（*）。



警告 不要用与函数或命令相同的名字给变量命名。如果你给了一个变量和某命令相同的名字，那么程序将无法运行；如果你给了一个变量和某函数相同的名字，那么在后面的程序中将无法使用相同的函数名，并且不会报错。

5.3 定义变量

变量使用之前必须先定义（define）。变量定义（有时称为变量声明，variable

declaration）是为了让C程序知道你需要一些变量空间，于是C程序会为你保留。定义一个变量，只需声明它的类型，后面跟上变量名。下面是某程序的第一行，这里定义了一些变量：

```
main()
{
   char initial;
   int age;
   float amount;
   /* Rest of program would follow */
```

上面的范例代码中有3个变量——initial、age和amount。它们可以存放3种不同类型的数据——字符数据、整型数据和浮点型数据。如果程序没有定义这些变量，就不能在变量中存放数据。

在同一行上可以定义多个相同类型的变量。例如，如果想定义两个字符变量而不是一个，那么你可以这样做：

```
main()
{
   char initial1, initial2;   /* Defines 2 characters */
   int age;
   float amount;
   /* Rest of program would follow */
```

或者这样做：

```
main()
{
   char initial1;
   char initial2;   /* Defines a second */
   int age;
   float amount;
   /* Rest of program would follow */
```



请跳过这里，这有点儿难　大多数C语言变量被定义在左大括号之后，如跟在函数名后面的左大括号。这些变量称为局部变量（local variable）。C语言也允许通过在函数名之前定义变量来创建全局（global）变量，如在main()之前。局部变量几乎总是比全局变量更可取。第30章会讲解局部变量和全局变量的区别，但目前，所有的程序都将用局部变量。

5.4　在变量中存储数据

赋值运算符（assignment operator）在变量中存放值。它用起来比听起来简单得

多。赋值运算符只是一个等号（=）。给变量赋值的格式如下：

```
variable = data;
```

其中，*variable*是想存放数据的变量名。你必须已经在前面定义了这个变量（像前一节所说的那样）。*Data*可以是数字、字符或产生数字的数学表达式。下面是3条赋值语句的范例，它们给前一节定义的变量赋值：

```
initial = 'G';  /* Assigns values to three variables */
age = 31;
amount = 2983.43;
```

你也可以在变量中存放表达式的值：

```
sales = 4432.67 / 1.20;  /* Divides to get value */
```

甚至可以在表达式中使用其他变量：

```
newSales = sales + 2167.65;  /* Uses value from
                                another variable */
```

图5-1说明了当你使用赋值语句时，C语言做了什么。



线索 等号会告诉C程序把等号右边的任何东西粘到左边的变量中。等号的作用就像往左指的箭头，表示“往这儿走”。哦，绝对不要在数字中使用逗号，无论数字有多大！

假设一个小企业支付你很多钱让你帮他们编写一个薪水计算程序。你就要定义一些浮点数来存放每小时支付的费用、工作的小时数以及税率等。附录B中的21点程序必须记录很多东西，因此用了很多变量。大多数程序会在函数的开始处定义变量。

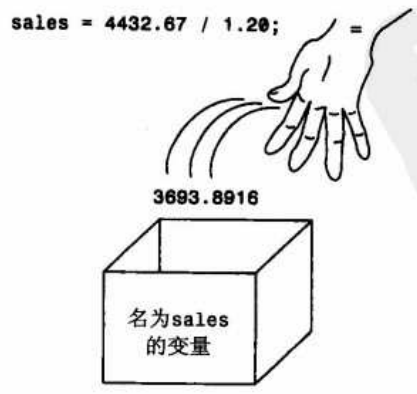


图5-1 赋值运算符把值放进变量中



线索　你可以在定义变量的同时给它赋初值。下面的代码定义了3个变量，并给其中两个（val和numSold）赋了初值：

```
int numSold = 25, numBought;
float val = 436.54;
```



奖励

- 学会如何命名变量，因为你将会在程序中用到它们。
- 总是在使用之前定义变量。
- 在同一行上可以定义多个变量。
- 等号称为赋值运算符。赋值运算符用来在变量中存储值。



陷阱

- 不要混淆数据类型和变量类型。避免把一种数据类型的值存储到另一种数据类型的变量中。
- 尽管可以，但最好不要在函数的前面定义变量（这样的变量称为全局变量，如果不小心使用，容易导致错误），而应该在函数的左大括号之后定义变量。目前对你来说这些局部变量使用起来更安全。
- 不要在数字中用逗号。输入值3万应该是30000，而不是30,000。

5.5　小结

本章介绍了C语言中不同的变量类型。因为存在不同类型的数据，所以必须有不同类型的变量来存放它们。数据类型的选择取决于程序员。要小心选择变量的数据类型，确保此类型与要存放在变量中的值相匹配。

当较小的数据类型足够存放数据时，不要使用大的数据类型。当int足够用时使用long int不仅会影响效率，而且还会导致程序庞大且运行缓慢。

1. 代码范例

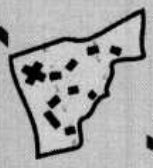
```
main()
{
  /* Defines a different variable for each data type */
  char priceCode = 'J';
  int quantity = 100;
```

```
long int wholeSaleQuant = 45000;
float price = 13.54;
double yrlySales = 9845543.23;
```

2. 代码分析

上面程序段中的语句既保留了变量的空间又给它们赋了初值。不过，并非总要给变量赋初值。很多时候你并不知道变量将存放什么值，因为这个值可能来自用户的键盘输入、磁盘文件，或者计算结果。无论是否知道初值，你都必须在使用它们之前为变量保留存储空间。

6



数组的每个单元
称为元素
51



如果你定义了数
组并对其初始化
51



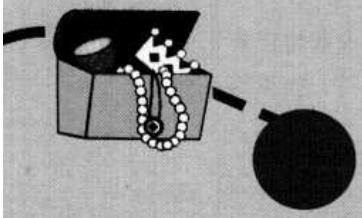
strcpy()会自动在字符串
末尾加上null零
52



ASCII表
48



别忘了数组的下标
从0开始
52



第6章

字 符 串

使用字符数组

本章内容

字符串结束符
48

字符串的长度
49

字符数组：字符的列表
49

初始化字符串
52

尽管C语言没有字符串变量，仍然有办法存储字符串数据。本章将讲解它的实现方法。你已经知道字符串数据必须放在引号中。即使是一个字符，如果放在引号中也是字符串。你也知道如何用`printf()`打印字符串。

剩下的唯一任务就是如何用一种特殊的字符变量类型来存放字符串数据，这样你的程序就可以输入、处理并输出字符串了。

6.1　字符串结束符

C语言对字符串做了最奇怪的事：在每个字符串的末端加上了一个零。字符串末端的零有几种名字。下面列出了一些：

- Null零
- 二进制零
- 字符串结束符
- ASCII零
- `\0`



警告　你唯独不能把字符串末端的零叫做零（zero）。C程序员给字符串末端的零起了特殊的名字，这样你就知道在字符串末端用的既不是普通的数值零也不是字符`'0'`。只有null零出现在字符串末端。

C语言用字符串末端的零标识所有字符串的结束。当你输入文字`My name is Julie`时，不需要做什么特别的事情。C程序会自动加上null零。你永远看不到null零，但它确实存在。在内存中，只有当C程序遇到null零时才知道字符串结束了。



说明　查看附录C时，你会找到ASCII表（在第2章中讨论）。第1个条目就是null，它的ASCII数是0。往下找到ASCII 48，会看到一个0。ASCII 48是字符`'0'`，而第1个ASCII值是null零。C语言把null零放在字符串的末端。即使字符串`"I am 20"`也以ASCII 0结束，它就在20的字符0后面。



线索　字符串结束符有时被称为`\0`（反斜杠零），这是因为可以把`\0`放在单引号中来表示null零。因此，`'0'`是字符零，而`'\0'`是字符串结束符。

图6-1展示了字符串"Crazy"在内存中是如何存储的。如你所见，尽管字符串只有5个字母，但它占用了6个字节（一个字节是一个存储单元）。作为字符串"Crazy"的一部分，null零也占用了一个字节。

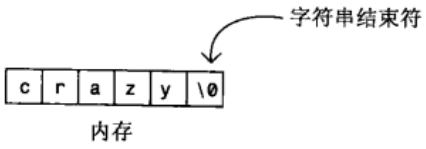


图6-1 在内存中字符串总是以null零结束

6.2 字符串的长度

字符串的长度（length）是指最大的字符数，但不包括null零。有时候你需要找出字符串的长度。计算字符串长度时，null零不能算在内。尽管字符串必须以null零结束（这样C语言知道字符串在哪里结束），但是null零并不是字符串长度的一部分。

根据定义，下面的字符串长度都是9个字符：

```
August 10
```

和

```
Batter up
```



警告 第1个字符串的长度不在10中的0处结束，因为10中的0不是null零，而是字符零。



线索 数据的所有单个字符的长度都是1。因此，'x'和"x"的长度都是1，但"x"占用了两个字符的内存，因为它带有null零。无论何时，每次看到引号中的字符串字面量时，就要想到在内存中字符串末端的null零。

6.3 字符数组：字符的列表

字符数组（character array）在内存中存放字符串。数组（array）是一种特殊的

变量类型，在后面的章节中你会了解更多。所有的数据类型——int、float、char等，都有相应的数组类型。数组只不过是具有相同数据类型的许多变量的一个列表。

在使用字符数组存放字符串之前，你必须告诉C语言你需要一个字符数组，也是在声明其他类型的变量的地方声明。在数组名后面使用方括号[和]，还要用一个数字来表明将要存放的字符串的最大字符数。

趣闻轶事
可以用C语言创建自己的计算机语言。

例子总是胜过千言万语。如果需要存放月份名字，你可以像下面这样定义一个名为month的字符数组：

```
char month[10];  /* Defines a character array */
```



线索 数组的定义很简单。去掉10和方括号，就是一个普通的字符变量。加上带有10的方括号就是告诉C语言你需要10个字符变量，它们都排列在名为month的列表中。

在定义该数组时用10的原因是最长的月份名September有9个字符。第10个字符是为——你已经猜到了——null零准备的。



线索 你总是得预留足够的字符数组空间容纳最长的字符串，还要算上字符串结束符。你可以定义比需要的更多的数组字符，但是不能比所需要的少。

如果你愿意，可以在定义数组的同时存储字符串：

```
char month[10] = "January";  /* Defines a character array */
```

图6-2展示了数组在内存中是如何存储的。因为数组的最后两位没有存放东西（January占用7个字符加上第8个字符null零），你不知道最后两位里放的是什么。但是，有些编译器会将未使用的元素用0填充。

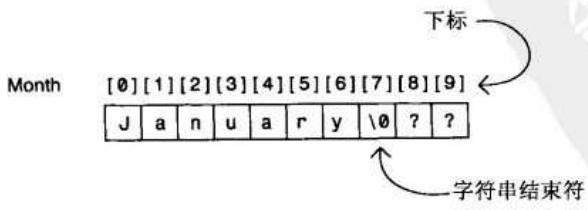


图6-2　定义和初始化名为month的数组，它用来存放字符串数据



线索 数组的每个单元被称为元素（element）。month数组有10个元素。你可以用下标（subscript）来区分它们。下标是你在方括号中指定的数字，可以访问到每个数组元素。

所有的数组下标以0开始。如图6-2所示，month数组中的第一个元素叫做month[0]。最后一个元素叫做month[9]，因为一共有10个元素，当从0开始时，最后一个就是9。

字符数组的每个元素都是一个字符。字符的组合——数组或字符列表——存放整个字符串。如果你愿意，可以一次一个元素地把数组的内容从January改为March，像这样：

```
month[0] = 'M';
month[1] = 'a';
month[2] = 'r';
month[3] = 'c';
month[4] = 'h';
month[5] = '\0';  /* Very important */
```

在字符串的末尾加上null零是至关重要的。否则，month数组仍然在month[7]处有一个null零。打印字符串时，你将得到：

```
Marchry
```



线索 打印数组中的字符串很简单。可以使用%s转化字符：

```
printf("The month is %s", month);
```



请跳过这里，这有点儿难 如果你定义一个数组并同时对其进行了初始化，你就可以不必在方括号里加上数字。下面的两句话做了相同的事：

```
char month[8] = "January";
```

和

```
char month[ ] = "January";
```

在第二个例子中，C语言计算January的字符数，并为最后一个null零加上1。但是，你就不能存放8个以上字符的字符串了。如果你想定义一个字符串的字符数组，并对其进行初始化，但又要为将来存放更长的字符串留下空间，你可这样做：

```
char month[25] = "January";     /* Leaves  room for
                                   longer strings */
```

6.4　初始化字符串

你一定不希望像前一节那样一次一个字符地初始化字符串。但是，与普通的非数组变量不同，你不能像下面这样把一个字符串赋给一个数组：

```
month = "April";  /* NOT allowed */
```

你只能在定义字符串的同时用等号给字符串赋值。如果在后面的程序中你想给数组一个新的字符串，就必须一次一个字符地赋值，或者用C编译器带的strcpy()函数（字符串复制）来实现。下面的语句把一个新的字符串赋给month：

```
strcpy(month, "April");  /* Puts new string in month array */
```



线索　在使用了strcpy()的程序中，必须在#include <stdio.h>后面加上这一行：

```
#include <string.h>
```



警告　不用担心。strcpy()会自动在字符串末尾加上null零。



说明　附录B中的21点游戏使用字符数组来存放玩家的名。看看你是否能找到使用字符数组的函数。可不要告诉我是main()函数。



奖励

- 在字符数组中存储字符串。
- 预留足够的数组元素容纳最长的字符串。
- 可以在定义字符数组时对其进行初始化，即一次给一个元素赋值，或者使用strcpy()函数实现。
- 如果使用strcpy()，要确保加上了#include<String.h>。



陷阱

- 只有当字符数组包含足够多的元素用于存放字符串时，才能把字符串放到字符数组中。
- 不要忘了数组的下标从0开始，而不是1（有些编程语言是从1开始）。
- 不要计算错了！如果你没有为结尾的零预留足够的元素，C语言将不能正确

处理字符数组。

6.5 小结

除了第5章中介绍的存储数值型数据，C语言还提供了存储字符串的方法。与单个字符的变量不同，字符串可以保存许多字符，如单词、地址和文本段。

C语言不支持字符串变量类型。尽管通过不支持字符串变量获得了一定的效率，程序员也不能放弃存储字符串数据的能力。因此，C语言提供了字符数组来以类似字符串的形式存放包含多个字符的数据。

1. 代码范例

```
/* Stores the days of the week in seven
different character arrays */
char day1[7] = "Sunday";
char day2[7] = "Monday";
char day3[8] = "Tuesday";
char day4[10] = "Wednesday";
char day5[] = "Thursday";
char day6[] = "Friday";
char day7[] = "Saturday";
char myName[6];
strcpy(myName, "Julie");
```

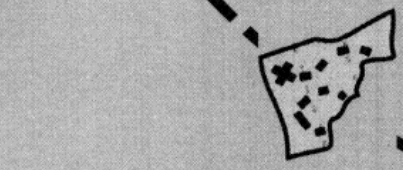
2. 代码分析

当你想存储字符串数据时，要像上面这样定义字符数组。你必须在变量名后面加上方括号，否则C语言会认为你在定义单个字符变量。记住，一定要为字符串末尾的null零预留足够的空间。如果在定义数组时给它赋了一个字符串，那么你就不必计算字符数并为null零又加1了。最后的3个day变量没有写上字符总数，因为C语言能够计算出要保存的数据所需的数目。

要注意上面的7个数组名正好都包含了数字，但是这些数字与下标没有关系。day4[]有10个元素，因为初始定义的下标为10。day4中的4只是为了与其他数组相区别。

如果你想定义没有赋值的数组，就像myName数组那样，那么必须在定义数组时包含最大的元素个数。然后可以用strcpy()给数组赋上字符串值。

7



你所包含的文件称为头文件
58



用大写字母定义常量
60



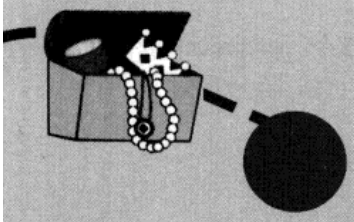
请保持C语言提供的头文件完好无损
58



编写自己的头文件
58



不要把定义常量当成变量
61



第7章

#include 和 #define

预处理器指令

本章内容

包含文件
56

在哪里放置#include指令
58

定义常量
59

在很多C程序中有两种代码行不是C命令。它们是预处理器指令（preprocessor directive）。预处理器指令总是以符号#开始，它们不会在运行时（即运行程序时）产生任何行为。它们只在编译程序时发生作用。

最常用的预处理器指令是：

- #include；
- #define。

在前面的章节中，你已经看到使用#include的例子。本章将最终揭开预处理器指令的神秘面纱。

7.1　包含文件

#include有两种格式，二者几乎等价：

```
#include <filename>
```

和

```
#include "filename"
```

图7-1展示了#include做了些什么。它只不过是一个文件合并（file merge）命令。就在程序被编译前，#include语句被#include后的文件的内容替换了。文件名可以用大写或小写字母表示，前提是你的操作系统允许这么做。例如，MS-DOS不区分文件名中的大写和小写字母，但UNIX却区分。如果你的文件名是myFile.txt，MS-DOS允许你使用下面任意一个#include指令：

```
#include <MYFILE.TXT>
#include <myfile.txt>
#include <myFile.txt>
```

但UNIX只允许你使用这个：

```
#include <myFile.txt>
```



线索　如果你用过文字处理器，你就很可能用过#include类型的命令。你可能曾将存储在硬盘上的文件合并到正在编辑的文件中。

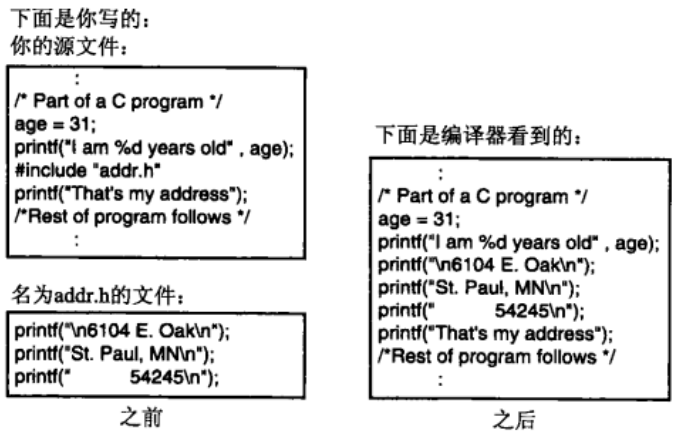


图7-1 #include把磁盘文件插入到另一个文件中

当你安装编译器时，安装程序会在硬盘上创建一个独立的位置（在一个目录中）存放编译器附带的各种#include文件。当你想要使用某个内置#include文件时，要使用带尖括号<和>的#include格式。



警告 你会问："我怎么知道什么时候要用内置的#include文件？"问得好！所有的内置函数，如printf()，都有相应的#include文件。本书在描述内置函数时，也会告诉你要包含哪个文件。



线索 编译器手册中有一个完整列表，上面列出了哪个函数包含在哪个文件中。



说明 你已经看到两个内置函数——printf()和strcpy()。main()不是内置函数，main()是一个你必须提供的函数。printf()对应的#include文件是STDIO.H（它代表了标准输入/输出），而strcpy()对应的#include文件是STRING.H。



线索 大多数C编译器提供上下文相关的帮助（context-sensitive help）。如果你的编译器有此功能，你可以将鼠标指针移到内置函数名（如strcpy()）上面，并选择上下文相关的帮助（一般可以按Alt+F1组合键）。编译器会在帮助消息中告诉你要使用这个函数必须包含哪个头文件。

本书中几乎所有的代码清单都包含如下的预处理器指令：

```
#include <stdio.h>
```

这是因为书中几乎每一个程序都使用了printf()。第6章会包含STRING.H，因为要讨论strcpy()函数。



线索　你所包含的文件称为头文件（header file）。这就是大多数被包含的文件以扩展名.H结束的原因。

如果你写了自己的头文件，就要用第二种形式的预处理器指令——带引号的那种。如果你使用引号，C语言会首先在程序所在的磁盘目录中搜索文件，再到内置的#include目录中搜索。由于这种搜索顺序，你可以编写自己的头文件，然后给它们与C语言内置的头文件相同的名字，这样你的文件就能代替C语言的文件使用了。



警告　如果你写了自己的头文件，不要把它们放在C语言内置的#include文件目录中。请保持C语言提供的头文件完好无损。几乎没有理由覆盖C语言的头文件，但你可以增加一些自己的头文件。



说明　当你有要在很多程序中经常使用的程序语句时，你或许可以编写自己的头文件。可把这些语句放在程序目录下的一个文件中，当你要用到它们时就用#include包含这个文件，这样就不用在每个程序中都输入它们了。

7.2　在哪里放置#include 指令

你用#include包含的头文件只是一些包含C代码的文本文件。后面你将学到更多关于头文件的知识，但现在只需理解头文件做两件事情。内置的头文件帮助C语言正确地执行内置函数。自己写的头文件通常包含你想要写到多个文件中的代码。



线索　最好把#include指令放在main()之前。



说明 附录B中的21点游戏程序包含了很多头文件，因为它使用了很多内置函数。注意#include的位置：它们出现在main()之前。

7.3 定义常量

#define预处理器指令用来定义常量（constant）。C常量其实就是字面量。在第2章中学习了字面量就是不会改变的值，就像数字4或字符串"C programming"。#define预处理器指令允许给字面量命名。当你给字面量命名时，命名后的字面量在C语言术语中称为命名常量（named constant）或定义常量（defined constant）。



警告 在第5章中你学过如何通过指定数据类型、变量名称和初始值来定义变量。用#define定义的常量不是变量，尽管它们在使用时有点像变量。

下面是#define指令的格式：

```
#define CONSTANT constantDefinition
```

如同C语言中大多数事物一样，使用定义常量实际上要比看起来的容易。下面是一些#define指令的例子：

```
#define AGELIMIT 21
#define MYNAME "Paula Holt"
#define PI 3.14159
```



线索 简而言之，#define告诉C：把程序中CONSTANT出现的每个地方都用constantDefinition替换。

上面的第一个#define语句指示C语言找出单词AGELIMIT出现的每个地方，并将其替换为21。因此，如果下面这条语句出现在#define后面的程序中：

```
if (employeeAge < AGELIMIT)
```

编译器会认为你输入了：

```
if (employeeAge < 21)
```

尽管你没有。



请跳过这里，这有点儿难　定义常量名要用大写字母。这是C语言中的一个少见的例外，而且推荐使用大写字母。因为定义常量不是变量，大写能使你在快速浏览程序时分辨出哪些是变量，哪些是常量。



线索　假设你前面定义过常量PI，大写字母能帮助你避免在程序中做这样的事情：

```
PI = 544.34;  /* Not allowed */
```

只要你坚持用大写字母定义常量，你就能知道不要去改变它们，因为它们是常量。

趣闻轶事
UNIX操作系统（DOS的竞争者）的大部分是用C语言写的。

有些值可能在程序运行期间要被修改，定义常量对于给这样的值命名是有好处的。例如，如果你不为AGELIMIT使用定义常量，而在程序中使用实际的年龄限制值如21，那么寻找并修改每个21是很困难的。如果你已经在程序顶部用了定义常量，当要修改年龄限制值时，你只需要像下面这样修改#define语句：

```
#define AGELIMIT 18
```



警告　#define指令不是C语言命令。与#include一样，C语言在编译程序之前处理#define语句。因此，如果你定义了PI为3.14159，并且在需要用到数学上的pi（π）的程序中使用了PI，那么当你在程序中实际上输入PI时，C语言会认为你输入了3.14159。PI更容易记住（帮助消除输入错误）且更清晰地表明为一个常量。

只要你在main()出现之前用#define定义了常量，则整个程序都能认得这个常量。因此，如果你在main()之前把PI定义为值3.14159，你就可以在main()和后面任何其他你写的函数中使用PI，并且编译器会在编译你的程序之前把PI替换为3.14159。



奖励

- 使用内置函数时需要使用正确的头文件。
- 当包含编译器提供的头文件时，用尖括号<和>把文件名括起来。
- 当包含位于源代码目录下自己写的头文件时，用引号"和"把文件名引起来。
- 在所有的定义常量中使用大写字母，这样你就能把它们与普通变量名区分开

来了。



陷阱

- 不要把为内置函数写的#include语句放在main()后面。在main()之前用#include包含头文件。你可以在任何时候用#include包含自己写的头文件。
- 不要把定义常量当成变量看待。与变量不同，一旦常量被定义之后，你就不能在常量中存储数据了。

7.4 小结

C语言的预处理器指令使C语言能够看到你没有实际输入的代码。例如，当你需要另一个文件（如STDIO.H）的内容时（此文件帮助你正确地产生输入和输出），你不必键入文件的内容。你只需通过#include指令指示C语言包含这个文件即可。

如果程序中有一个常量值，如销售奖金限制，你应该在程序的顶部用#define定义这个常量。在程序中就只需键入常量的名字，而不用键入实际数字了。如果这个限制值发生了改变，你只需要修改#define这一行代码。

#include和#define不是与程序的其他部分一起运行的C语句。它们只对源代码上起作用，即在C语言编译程序之前包含文本或者把定义名改为实际的值。

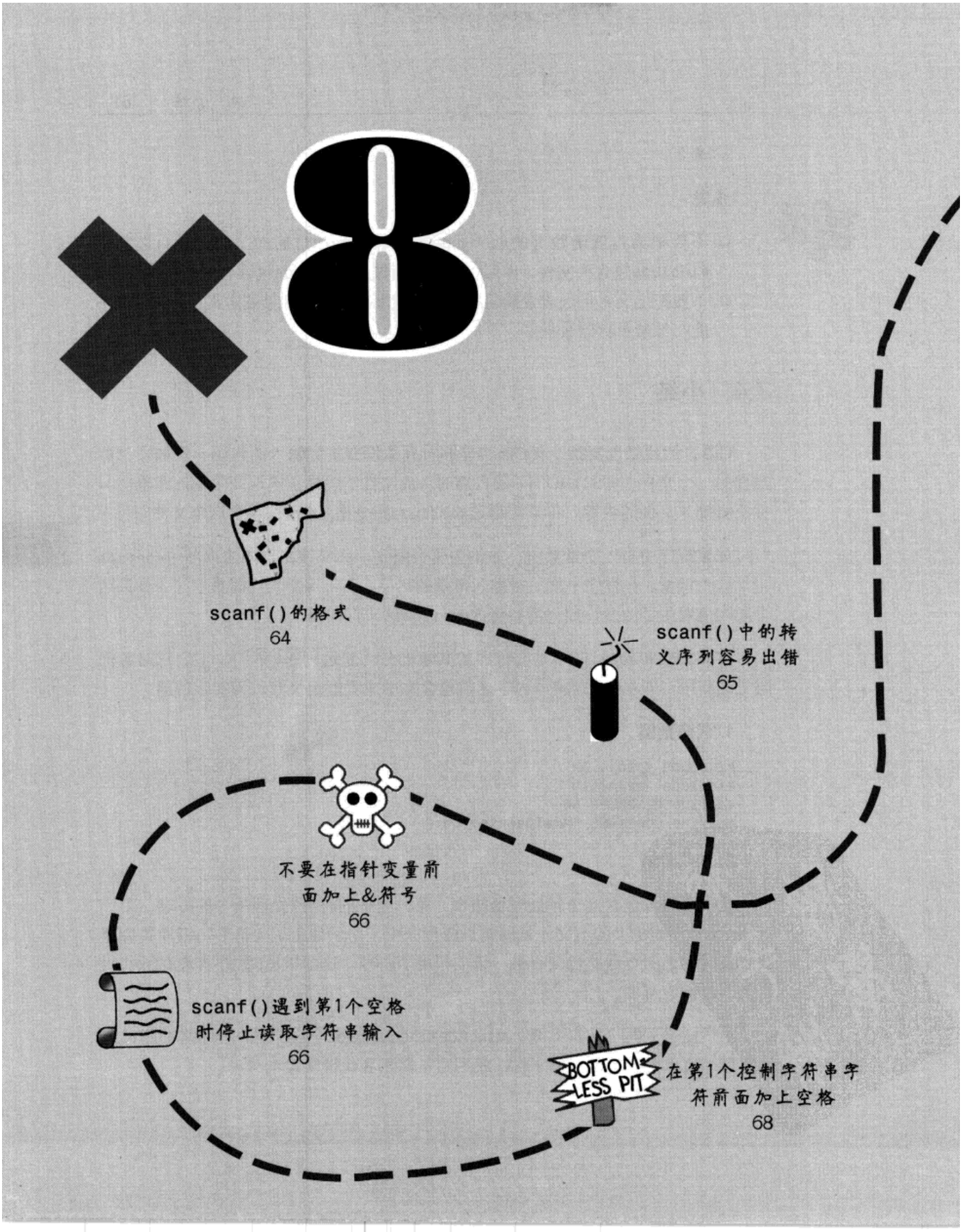
1. 代码范例

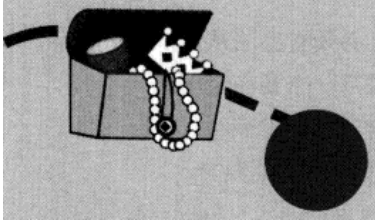
```
#include <stdio.h>
#include "mycode.h"
#define MINORDER 50
#define COMPNAME "Amalgamated Co."
```

2. 代码分析

这段代码只包含几个预处理器指令。第一列中的#是预处理指令的标志。第一行指示C程序把STDIO.H文件复制到当前程序中。由于使用了尖括号，编译器会查找安装时创建的普通的包含目录。第二行用了引号，这会指示C程序首先在源代码目录下查找该文件。

随后定义了两个定义常量，MINORDER和COMPNAME。当后面的程序需要测试或打印最小定单数量或公司名字时，就用定义常量名代替常量本身。


scanf()的格式
64
scanf()中的转义序列容易出错
65
不要在指针变量前面加上&符号
66
scanf()遇到第1个空格时停止读取字符串输入
66
BOTTOM LESS PIT
在第1个控制字符串字符前面加上空格
68



第8章

输　入

使用scanf()

本章内容

简述scanf()
64

与printf()一起使用
65

使用scanf()的问题
65

printf()把数据发送到屏幕，而scanf()函数从键盘获取数据。必须有办法从用户那里获取数据，不能总是使用赋值语句进行赋值。例如，如果你编写一个要在全国推广使用的录像带租用程序，你就不能在程序中用等号把带子的租金赋给一个变量，因为各家店的租金都不相同。你得在计算费用之前询问各家店里带子的租金是多少。

你可能不会喜欢学习scanf()，所以我们将尽可能快学完它。你将发现scanf()是最危险的函数。对于初学者来说，scanf()不太好理解，它有着奇怪的格式，但是由于它与printf()函数有紧密联系，到本书目前为止它还是最容易用于输入的。

8.1　简述 scanf()

图8-1展示了scanf()做了些什么。scanf()是所有C编译器都带有的内置C函数。它的头文件与printf()相同——stdio.h——所以你不需要担心包含其他头文件。

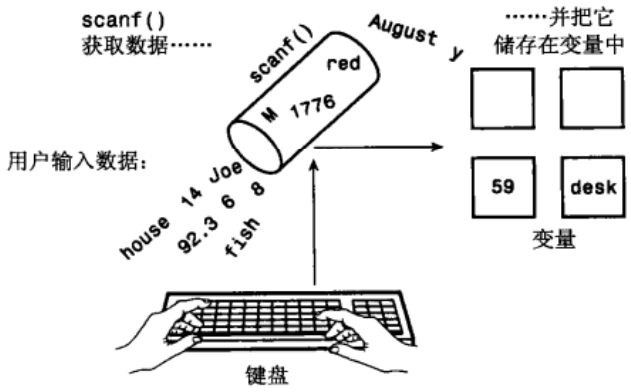


图8-1　scanf()把用户输入的值填到变量中



说明　如果你掌握了printf()，scanf()就十分简单了。scanf()看起来与printf()很像，因为scanf()也用到了转化字符如%s和%d。



线索　scanf()是printf()的镜像函数。你写程序会经常用printf()向用户请求值，再用scanf()获取值。下面是scanf()的格式：

```
scanf(controlString [, data]);
```

程序执行到scanf()时，C程序会停下来并等待用户输入数据。在scanf()内列出的变量（在controlString后面）将接受用户输入的任何值。当用户在输入之后按下Enter键时，scanf()退出。



请跳过这里，这有点儿难 尽管scanf()使用了和printf()相同的转化字符，但是不要用转义序列如\n、\a或\t。转义序列会使scanf()混乱。当用户按下Enter键时，scanf()停止从用户读取数据，所以不能用\n。

8.2 与 printf()一起使用

几乎每个scanf()前面都要有printf()。如果没有printf()，程序会停下并等待输入，而此时用户并不知道该做什么。例如，如果需要从用户获取一个数量，你可以在scanf()前面加上这样一句printf()：

```
printf("What is the amount? ");  /* Prompt */
/* A scanf() would follow */
```



线索 scanf()前面的printf()给了用户一个提示。如果你不提示用户需要的值，用户就不知道应该输入什么值。通常，printf()向用户请求数据，scanf()获取用户输入的数据。

8.3 使用 scanf()的问题

如本章第一部分所提到的，scanf()不是最易用的函数。scanf()一开始碰到的问题就是，尽管用户必须准确地按要求输入数据，但是他们经常做不到！如果scanf()需要一个浮点值，但是用户却输入了一个字符，那就会一事无成。由于字符不是浮点值，所以提供的浮点变量中存放了错误的值。

趣闻轶事 数据输入是用于获取用户输入的术语。

现在，假设用户确实输入了所需的数据。第18章介绍了一些方法克服scanf()带来的问题（现代的C程序员也经常借助于自己编写、下载或购买的完整的数据输入程序，这些程序可以克服C语言难以获取数据输入的难题）。



请跳过这里，这有点儿难 你可能会问："我还没有看到scanf()。什么时候给我展示一个？"请做好准备，下面就是了。

下面的scanf()从键盘读取一个整型值（可以根据%d分辨出来）到变量age中：

```
scanf(" %d", &age);
```

变量age将存放用户在按下Enter前输入的任何值。

关于scanf()，要注意的第一点是%d前面的空格。这里的空格不是必需的，但它没有坏处，有时候当你连续地获取数字或字符时，它能帮助你更好地输入。在学习scanf()时记住添加空格是一个好习惯。

下面来看看最明显的scanf()问题——在变量age前面的&符号。scanf()要求在所有变量前面加上&符号，尽管&不是变量名的一部分。如果这样做了，scanf()就能工作；去掉&，scanf()就不能把用户输入的值存到变量中。



线索　有一个&规则的例外你应该知道。如果用%s把输入存到一个数组中，就像当你把用户输入的名字存储到字符数组那样，就不要用&符号了。

基本规则是：如果要求用户输入整数、浮点数、字符、双精度数或其他任何一个变量的组合（长整数等），就要在scanf()中的变量名前面加上&。如果要求用户输入字符串到一个字符数组中，就不要在数组名前面加上&。



警告　你也不能把&放在指针（pointer）变量前面。实际上，数组只不过是一个指针变量，这就是数组不需要&的原因。本书后面将介绍指针，但是如果你已经在其他语言中见过指针变量，你就知道我在说什么了。否则，你很难理解这里在说什么，就别再读本段了。在学完第25章之后，你将完全理解指针以及它与数组的相似性。



说明　关于用scanf()把字符串读到字符数组中，你还需要知道一个问题。scanf()在遇到第1个空格时停止读取字符串输入。因此，用scanf()只能一次读取一个单词。如果你必须读取用户的多个单词，如用户的名和姓，就得用两个scanf()（带有各自的printf()提示）并存放到两个字符数组中。

下面的程序要求用户输入名、姓、年龄和体重。注意，字符数组前没有&，但其他两个变量前面有。由于scanf()不能一次处理两个单词，所以此程序没有要求

用户输入全名。

```
#include <stdio.h>
main()
{
  int age;
  float weight;
  char first[15], last[15];

  printf("\nWhat is your first name? ");     /* A prompt */
  scanf(" %s", first); /* No ampersand on character arrays */
  printf("What is your last name? ");
  scanf(" %s", last);  /* No ampersand on character arrays */

  printf("How old are you? ");
  scanf(" %d", &age);  /* Ampersand required */
  printf("How much do you weigh? ");
  scanf(" %f", &weight);

  printf("\nHere is the information you entered:\n");
  printf("Name: %s %s\n", first, last);
  printf("Weight: %.0f\n", weight);  /* 0 decimal places */
                                     /* displayed        */
  printf("Age: %d", age);
  return 0; /* Always best to use this. I'll explain later */
}
```

下面是些程序的执行示例：

```
What is your first name? Joe
What is your last name? Harrison
How old are you? 41
How much do you weigh? 205

Here is the information you entered:
Name: Joe Harrison
Weight: 205
Age: 41
```



线索 你可以让用户输入数据值之外的其他字符。例如，日期在输入的时候经常以斜杠或连字符来分隔年、月、日，像这样：03/05/95。你得相信用户会输入正确的数据。下面的scanf()获取日期，并且要求用户以mm/dd/yy格式输入：

```
scanf(" %d/%d/%d", &month, &day, &year);
```

用户可以输入02/28/94或11/22/95，但不能输入June 5th, 1993，因为scanf()要求用户以mm/dd/yy格式输入。



奖励

- 如果想通过键盘从用户获取数据，就使用scanf()。
- 每个scanf()都需要一个控制字符串，它表明输入时数据的格式。
- 在用scanf()之前，最好要用printf()提示用户要输入的值。
- 在scanf()中的非数组变量之前要加上&。



陷阱

- 不要忘记在第1个控制字符串字符前面加上空格（例如，"%d"在%前有一个空格）以确保正确地输入。
- 不要在scanf()的数组名前面加上&。
- 不要期望用户精确地输入你想要的东西！如果需要精确性，比如非计算机专业人员在终端用户环境下使用你的程序，你就要用第18章介绍的其他输入方法。

8.4　小结

本章教你如何向用户提问并获取答案。能够处理用户输入对任何语言来说都是很重要的一部分。scanf()执行数据输入，也就是说，scanf()获取用户的输入并存储在变量中。

在使用scanf()之前，需要让用户知道该输入什么。例如，在用scanf()获取销售数量之前，用printf()询问数量，于是用户就知道该输入什么了。

尽管scanf()使用和printf()相同的代码格式，但是scanf()还有另外的要求。scanf()要求在非数组变量前加上&。另外，如果你从用户获取字符串，scanf()和%s组合每次只能得到一个单词。

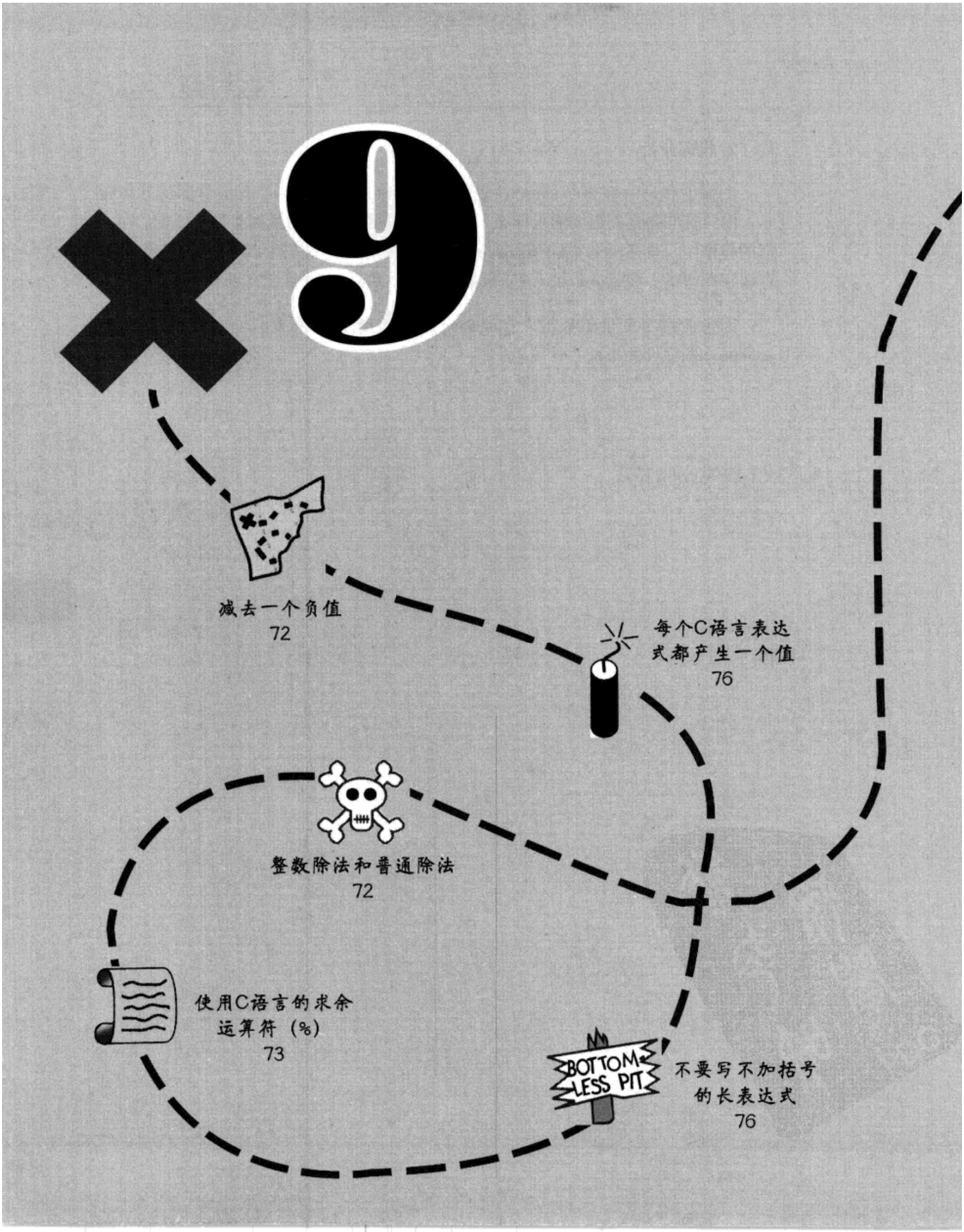
1. 代码范例

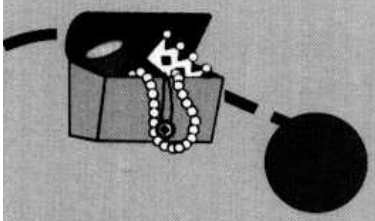
```
/* Asks users for their hometown, state, and year born */
printf("What town were you born in? ");
scanf(" %s", homeTown);  /* homeTown must be a
                            character array */
printf("What state were you born in? ");
scanf(" %s", state);  /* state must be a character array */
printf("What year were you born in? ");
scanf(" %d", &yearBorn);  /* yearBorn must be numeric */
```

2. 代码分析

上面的代码行在每个scanf()之前用printf()提示用户。有了前面的打印提示，用户就准确地知道要输入什么。代码格式必须与变量的数据类型相匹配。在之前的程序中，变量homeTown和state必须已经被定义成字符数组，因为它们是用来存放字符串的。变量yearBorn是int数据类型。用int足够存放年份的值了。

因为前两个变量是数组，在scanf()中它们前面不需要&。在非数组变量如yearBorn前面要加上&。


减去一个负值
72
每个C语言表达式都产生一个值
76
整数除法和普通除法
72
使用C语言的求余运算符（%）
73
BOTTOM LESS PIT
不要写不加括号的长表达式
76



第9章

C怎么做数学运算

使用运算符

本章内容

基本知识
72

运算符的优先级
74

用括号打破规则
75

多重赋值
75

单词operator有两种意思。一种意思是电话接线员，他们帮助你接通长途电话，但这里不讨论他们。另一种意思是C语言的运算符，用来帮你做数学运算。不用成为数学天才就能使用这些数学运算符编写程序。只要你知道如何正确地排列运算符，C语言就会做所有的数学计算。

你不仅要学会识别数学运算符，还要懂得这些运算符的优先级。C语言总是从左到右进行计算。本章将一一讲解。

9.1　基本知识

许多C运算符都可以按照你所期望的方式工作。想做加法时就用加号（+），想做减法时就用减号（-）。一个表达式（expression）包含了一个或多个运算符。当给变量赋值时，C程序员经常在赋值运算符右边使用数学表达式，像这样：

```
totalSales = localSales + foreignSales - salesReturns;
```

C程序会计算出结果并将其存储到totalSales中。



线索　如果想减去一个负值，要确保在两个减号之间加上空格，像这样：

```
newValue = oldValue - -factor;
```

如果你省略了空格，C语言会认为你使用了另一个运算符--，它称为递减（decrement）运算符，将在第13章中介绍。

你甚至可以把数学表达式写在printf()中：

```
printf("In 3 years, I'll be %d years old.\n", age + 3);
```

如果想做乘法和除法，可以用*和/。下面的语句用乘法和除法给变量赋值：

```
newFactor = fact * 1.2 / 0.5;
```



警告　如果在除号（/）两边都是整数，C语言会计算出整数除法的结果（integer division result）。学习下面的语句，熟悉整数除法和普通除法。注释解释了除法得到的结果：

```
float a = 17.0;
float b = 5.0;
float answf;    /* Holds float answer */
```

```
int i = 17;
int j = 5;
int answi;      /* Holds integer answer */
answf = a / b;  /* 3.4 is stored in answf */
answi = i / j;  /* 3 is stored in answi */
```



说明 如果你需要整数除法得到的余数，可以使用C语言的求余（modulus）运算符（%）。给定上面列出的值，下面的语句把2赋给ansMod:

```
ansMod = i % j;  /* 2 is the remainder of 17 / 5 */
```

现在你知道了C语言的3种除法：如果/的一边或两边是非整数就执行普通除法，如果/的两边都是整数就执行整数除法，如果对两个整数用了%运算符就执行求余运算。



线索 %运算符只能用于整型数据。

下面的短程序计算轮胎的净销售价格：

```
/* Computes net tire sales */
#include <stdio.h>
main()
{
  int numTires;
  float tirePrice, beforeTax, netSale;
  float taxRate = .07;      /* Sales tax */
  /* Ask user for values */
  printf("How many tires bought? ");
  scanf(" %d", &numTires);
  printf("How much is each tire? ");
  scanf(" %f", &tirePrice);
  /* Compute the sale */
  beforeTax = tirePrice * numTires;
  netSale = beforeTax + (beforeTax * taxRate);
  printf("Your total sale is %.2f\n", netSale);
  return 0;
}
```

下面是程序运行的示例：

```
How many tires bought? 4
How much is each tire? 45.99
You total sale is 196.84
```

9.2　运算符的优先级

如本章前面所述，C语言不会总是按照你期望的顺序做数学运算。下面的表达式简要地说明了这一点：

```
ans = 5 + 2 * 3;  /* Puts 11 in ans */
```

如果你认为C语言会在ans中存储21，那么你是从左到右地读表达式。然而，C语言的乘法运算总是优先于加法运算！这听起来不可思议，但只要你掌握了规则，你就会认可。C语言遵守运算符优先级（order of operator）表。C程序首先把2和3相乘得到6，再加上5得到11。

趣闻轶事
第1个计算机bug是一只真正的虫子。一台军用打印机不能正常工作了，结果在两根电线中发现了一只飞蛾。

你可以在本书最后可以撕下来的那张卡片页（tearout card）上找到完整的运算符优先级表。可以看到，*、/和%出现在+和-之前。因此，如果C语言遇到带有这些运算符组合的表达式，它会在计算+和-之前求*、/和%的值。

下面是一个较难的表达式。所有的变量和数字都是整数。看看你是否能按照C语言给表达式求值的方式计算出它的值：

```
ans = 5 + 2 * 4 / 2 % 3 + 10 - 3;  /* What is answer? */
```

答案是13，如图9-1所示。

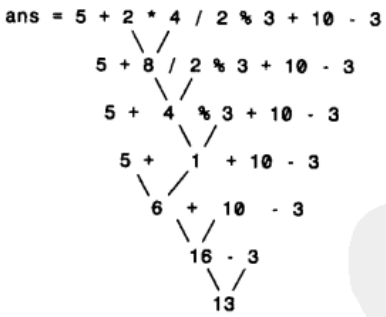


图9-1　按照C语言的方式求解表达式



线索　在练习求表达式的值时，不要一次求太多运算符的值。如图所示，你应该一次计算一个运算符，然后把表达式的余下部分留到下一轮。

如果一个表达式（例如图9-1中的）中包含了多个相同优先级的运算符，就必

须用运算符优先级表中的第3列（标有结合性）来判断如何给运算符求值。换句话说，因为*、/和%都处在同一级别上，它们按照运算符优先级表中的结合性列所指明的从左到右的顺序求值。

你可能想知道为什么得学习这些。毕竟，难道不是C语言为你做数学运算吗？回答是“是，但是……”C语言做了数学运算，但你要知道如何正确地书写表达式。举个简单的例子：假设你想计算4个变量的平均值，下面的写法是不行的：

```
avg = i + j + k + l / 4;  /* Will NOT compute average! */
```

一旦你理解了运算符的优先级，原因就很简单了。C语言首先计算除法，于是1/4首先被求值，然后i、j和k被加到除法的结果上。如果想改变运算符的优先级，你就得学会在表达式中使用括号。

9.3 用括号打破规则

如果需要，你也可以改变运算符的优先级。如果把一个表达式写在括号中，C语言就会优先对这个表达式求值。因为运算符优先级表把括号放在其他任何数学运算符之前，括号有优先权，如下面的语句所示：

```
ans = (5 + 2) * 3;  /* Puts 21 in ans */
```

尽管乘法通常在加法之前运算，但是括号迫使C语言先求出5+2的值再给结果7乘以3。因此，如果想求4个数的平均值，你可以通过把加法运算加上括号来实现：

```
avg = (i + j + k + l) / 4;  /* Computes average */
```



线索 请大量地使用括号。它们能使你的表达式更清晰。即使普通的运算符优先级就能够满足你的表达式，但使用括号可以使表达式更易于理解，方便日后的修改。

9.4 多重赋值

从运算符优先级表中可以看出，赋值运算符有优先级和关联性，其他运算符也有。赋值运算符在表中的优先级很低，并且它是从右到左关联的。

从右到左的关联性可以做有趣的操作：你可以在同一个表达式中把一个值赋给

多个变量。要把9赋给10个变量，你可以这样写：

```
a = 9; b = 9; c = 9; d = 9; e = 9;
f = 9; g = 9; h = 9; i = 9; j = 9;
```

但是下面这样写更简单：

```
a = b = c = d = e = f = g = h = i = j = 9;
```

由于从右到左的关联性，C语言首先把9赋给j，然后再把9赋给i，依此类推。



线索　C语言不会替你初始化变量。如果你想让很多变量的值为0，可以通过多重赋值来实现。



请跳过这里，这有点儿难　每个C语言表达式都会产生一个值。表达式j = 9;确实把9存放到j里，但是同时它也产生了一个值9，如果需要的话可以存放到其他地方。每个赋值都会产生一个表达式，使你可以做一些在其他编程语言中也许不能做的事，像下面这样：

```
a = 5 * (b = 2); /* Puts a 2 in b and a 10 in a */
```



奖励

- 用+、-、*和/进行加法、减法、乘法和除法运算。
- 如果想要得到整数除法的余数，就用%。
- 手边最好备有运算符优先级表，以方便查阅，因为它决定了C语言如何对表达式进行求值。
- 如果有几个变量需要初始化，就用多重赋值运算符。



陷阱

- 如果你想减去一个负数，不要把两个减号连在一起。在它们之间留个空格。
- 不要用%计算非整型数据除法的余数。如果除以非整数，得到的结果是精确的浮点数。
- 不要写不加括号的长表达式。括号有助于让表达式更加清晰、准确，避免由于遵循运算符优先级表而出错。

9.5 小结

C语言提供了很多的数学运算符进行计算。大多数运算符与它们对应的数学符号很像（+、-和/等），而其他的也容易学习。你必须要特别注意运算符的优先级。C语言的运算符优先级表展示了C语言理解计算顺序的方式。如果需要的话，可以用括号改变内置的顺序。

1. 代码范例

```
total = cost * numberBought;
sTax = .08 * total;
grandTotal = total + sTax;
discounted = grandTotal - .10 * grandTotal;
```

2. 代码分析

上面的一连串赋值语句计算出总的销售额和每次购物的打折金额。物品的单价首先乘上购买的数量。8%的税必须从总数中除去，然后再加回来。如果顾客付的是现金，要从总数之和中计算10%的折扣。

每一步计算都不需要用到括号，因为在这个例子中用正常的运算符顺序就够了。如果你想让计算顺序更清晰，可以加上括号，最后一句话也可以这样写：

```
discounted = grandTotal - (.10 * grandTotal);
```



Part 2 第二部分

操作空间

本部分内容

- 第 10 章　表达式还能用来做什么
- 第 11 章　关系运算符
- 第 12 章　逻辑运算符
- 第 13 章　更高级的运算符



10



你可能给变量定义了错误的类型
85



可以对整个表达式进行强制类型转换
85



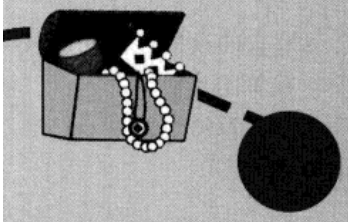
数学上，没有一个数等于它自己加上1
82



复合运算符能减少输入错误
83



不要混用不同的数据类型
86



第 10 章

表达式还能用来做什么

复合运算符和强制类型转换

本章内容

复合赋值
82

小心优先级
84

强制类型转换
85

从运算符优先级表中可以看出，C语言有多种多样的运算符。C语言的众多运算符有助于让C语言命令更加简练。C语言中的命令不是很多，但它有着比其他大多数编程语言多得多的运算符。大多数计算机编程语言有着相对较少的运算符和较多的命令，C语言通过提供许多功能很强的运算符来保持其简练的特点。

本章将讲述C程序要用到的其他一些运算符。具体将讲解复合赋值和强制类型转换，以便在编写高级程序时使用。

10.1　复合赋值

在程序中，你经常不得不修改变量的值。到目前为止，所有的变量都是基于常量字面值或表达式进行赋值。然而，你也经常需要更新变量的值。

假设程序需要计算利润值小于0的次数。你需要设置一个计数器变量（counter variable）。计数器变量是指这样的变量：当特定事件发生时，它的值增加1。每当利润值为负数时，则执行下面的语句：

```
lossCount = lossCount + 1;   /* Adds 1 to lossCount variable */
```



警告　在数学上，任何东西都不等于自身加1！在计算机中，上面的赋值语句给lossCount加上1，然后把新值赋给lossCount——效果是给lossCount的值加1。请记住等号的意思是：将等号右边的任何东西取出，并将计算出的值存储到左边的变量中。

下面的程序很简单，它通过在每个printf()前面用计数器赋值语句来打印出数字1到5：

```
/* Counts from 1 to 5 */
#include <stdio.h>
main()
{
  int ctr = 0;      /* Start at 0 */
  ctr = ctr + 1;    /* Add 1 to counter */
  printf("ctr is at %d\n", ctr);
  ctr = ctr + 1;    /* Counter is now 2 */
  printf("ctr is at %d\n", ctr);
  ctr = ctr + 1;    /* Counter is now 3 */
  printf("ctr is at %d\n", ctr);
  ctr = ctr + 1;    /* Counter is now 4 */
  printf("ctr is at %d\n", ctr);
  ctr = ctr + 1;    /* Counter is now 5 */
  printf("ctr is at %d\n", ctr);
  return 0;
}
```

下面给出了程序的输出。请注意每条赋值语句使ctr增1（用计算机行话来说，叫递增）。

```
ctr is at 1
ctr is at 2
ctr is at 3
ctr is at 4
ctr is at 5
```

通过用减法代替加法，你可以用类似的方式递减（decrement）变量的值（比如当产品被卖掉时想减少货物总量）。

还有其他一些情况你需要更新（update）变量，或者给总数加上值或者以某种方式调整它。下面的赋值语句给变量sales的值增加25%：

```
sales = sales * 1.25;  /* Increases sales by 25 percent */
```

C语言提供了几种复合运算符（compound operator），它们使你以与前面描述的方法（递增、递减和其他更新）相似的方式更新变量的值。然而，不用在等号两边都重复变量名，你只需在一边列出一次变量。我们再通过一些例子来更好地弄清楚复合运算符。



线索 第15章将会展示for语句如何使更新变量变得更容易。

如果想给某变量加1，可以使用复合加法运算符+=（compound addition operator）。下面两条语句将会产生相同的计算结果。

```
lossCount = lossCount + 1;  /* Adds 1 to lossCount variable */
```

和

```
lossCount += 1;  /* Adds 1 to lossCount variable */
```

不用再像下面这样给sales乘上1.25再把计算结果赋给它自己：

```
sales = sales * 1.25;  /* Increases sales by 25 percent */
```

你可以用复合乘法运算符*=（compound multiplication operator）来实现：

```
sales *= 1.25;  /* Increases sales by 25 percent */
```



说明 使用复合运算符要比写出整个赋值语句更快，因为你不需要在等号两边都写出相同的变量名。另外，复合运算符也减少了输入错误，因为你不用在同一语句中两次输入相同的变量名。

表10-1列出了所有的复合赋值运算符并给出相应的例子。到目前为止你在本书中见过的所有运算符，从加法到求余，都有对应的复合运算符。

表10-1　复合赋值运算符

复合运算符	例　子	等价的语句
*=	total *= 1.25;	total = total * 1.25;
/=	amt /= factor;	amt = amt / factor;
%=	days %= 3;	days = days % 3;
+=	count +=1;	count = count + 1;
-=	adjust -= 0.5;	adjust = adjust - 0.5;



说明　在附录B中，21点游戏中的dispCard()函数根据最后抽取的纸牌用复合加法运算符更新纸牌数目。

10.2　小心优先级

查看运算符优先级表并确定复合赋值运算符所处的位置，你会看到它们的优先级很低。例如，+=比+低好几级。

起初，这听起来可能不是件大事。实际上，这里可能没有任何东西听起来像大事。如果真是这样，就太好了！C语言原本就不难。运算符优先级表会使粗心的程序员困窘。你会如何对下面的第二条语句求值：

```
total = 5;
total *= 2 + 3;  /* Updates the total variable */
```

第一眼看上去，你可能会认为total的值是13，因为在前面已经学过乘法比加法先算。乘法比加法先算这一点是对的，但是根据运算符优先级，复合乘法（compound multiplication）要比加法后算。因此，2+3得到5，然后5再乘以total的旧值（正好也是5）得到25，如图10-1所示。

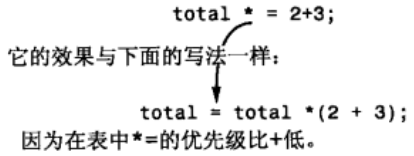


图10-1　复合运算符的优先级较低

10.3 强制类型转换

单词typecasting有两种意思：一种是电影导演经常要做的事（但不在这里讨论），另一种是C语言中的强制类型转换。C语言中的强制类型转换（typecast）是暂时将变量的数据类型改变为另一种。下面是强制类型转换的格式：

```
(dataType)value
```

*dataType*可以是任何C语言中的数据类型，如int或float。*value*是任何变量、字面量或表达式。假设age是一个值为6的整型变量，下面的语句：

```
(float)age
```

把age转换成值为6.0的float类型。如果你在含有其他float数据的表达式中使用age，就应该把age转换为float类型以保持一致性。



说明 由于混用数据类型会自动产生一些舍入问题，所以如果显式地把表达式中的变量和字面量都转换成同一数据类型，问题就会少很多。

不要一味地对每一行的变量进行强制类型转换。在不得不对变量或表达式进行转换的地方进行强制类型转换，才能得到正确的结果。前面对age的强制类型转换也可以表示成这样：

```
salaryBonus = salary * (float)age / 150.0;
```

age没有变为浮点型变量！age只是在这次计算中被暂时地（temporarily）改变。在程序中的任何地方，只要age没有被显式地转换，它就仍然是int类型变量。



线索 如果你发现整个程序中都把同一个变量转换成另一种类型，那么很可能在开始的时候就给变量定义了错误的类型。



请跳过这里，这有点儿难 你可对整个表达式进行强制类型转换。下面的语句在把表达式的结果赋给一个变量之前对它进行了强制类型转换：

```
value = (float)(number - 10 * yrsService);
```

表达式两边的括号避免只对变量number进行强制类型转换。C语言确实会做一些自动的强制类型转换。如果value定义为float类型，C语言会在把结果存储到value中之前，对上面的表达式进行强制类型转换。然而，如果你想使所有的表达式清晰明了并且不依赖于自动类型转换，就要对表达式进行强制类型转换了。



奖励

- 在更新变量值时用复合赋值运算符。
- 使用复合赋值运算符可以消除一些输入错误，并且减少程序的编写时间。
- 在变量、表达式或数据值前面加上带括号的数据类型，可以对其进行强制类型转换。



陷阱

- 不要混用不同的数据类型。要对数据类型进行强制转换，使得在求值之前所有数据的类型一致。
- 不要忽视运算符的优先级！在表中，复合运算符的优先级较低，而且在几乎其他所有运算符计算结束后才开始操作。

10.4　小结

本章介绍了C程序要用到的其他一些运算符。当你想修改变量的值时，可以使用复合运算符。你不必在等号两边都重复写变量名，因为复合运算符明白你想更新变量的值。这就减少了很多输入的潜在错误，特别是使用像`costOfGoodsSold`这样的长变量名时。

如果你想混用不同数据类型的变量和常量，请用强制类型转换来产生一致的数据类型。尽管C语言会自动进行强制类型转换，但是结果可能不是你想要的。当你想让所有数据都是`float`类型时，C语言可能会把`float`转换成`int`。后面编写使用指针和结构的更高级的程序时，强制类型转换会变得至关重要。

1. 代码范例

```
/* Dog age calculation */
#include <stdio.h>
main()
{
  int age;
  float dogAge;
  printf("How old are you? ");
  scanf(" %d", &age);  /* Gets the user's age as an integer */
  dogAge = (float)age / 7.0;
  printf("If you were a dog, you'd only be %.1f years old!",
         dogAge);
  return 0;
}
```

2. 代码分析

上面的代码首先要求用户输入其年龄。因为人们极少输入带小数点的年龄，所以年龄的值存储在一个整型变量中。然后，年龄被`7.0`除，得到对应的狗的年龄。狗的1年相当于人的7年，你一定想知道灵犬莱茜[①]是怎样从20世纪50年代活到现在的。

因为除法产生浮点结果，强制类型转换确保`age`在计算之前被转换成`float`型。在编写高级程序时，保持数据类型的一致性非常重要。

① 在美国，灵犬莱茜（Lassie）是人人都知道和喜爱的狗明星。不论是在卡通、电视剧或者电影，都有它的踪影。——编者注

11



尽量使用大括号
92



else的主体……
93



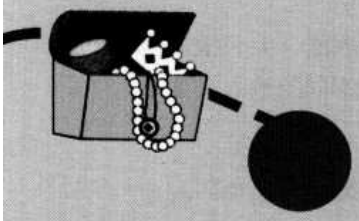
不要比较字符和
浮点数
91



关系运算符比
较两个值
90



不要在if或else
后面加分号
94



第 11 章

关系运算符

使用关系运算符

本章内容

测试数据
90

使用if语句
91

否则……：用else语句
92

C语言提供了一种极其有用的if语句。if语句使程序可以做判断，并且根据判断的结果执行特定的语句。通过测试变量的内容，程序可以根据不同的输入产生不同的输出。

本章还描述了*关系运算符*（relational operator）。把关系运算符和if语句一起使用，使C语言成为一种强大的数据处理语言。如果电脑不能测试数据的话，它们会令人感到厌烦。如果电脑没有根据数据决定行动方针的能力，那么它们就仅仅是计算器了。

11.1　测试数据

C语言的if语句的工作方式与在口语中的一样：*如果某事为真，就做一件事；否则，就做另外的事*。看一看下面的语句：

如果我挣了足够多的钱，我就去意大利。

如果鞋子不合适，就把它们拿回来。

如果外面很热，就给草地浇水。

表11-1列出了C语言的关系运算符，它们可以用来测试数据。请注意，有些关系运算符是由两个符号构成的。

表11-1　C语言的关系运算符

关系运算符	描　述
==	等于
>	大于
<	小于
>=	大于等于
<=	小于等于
!=	不等于



说明　*关系运算符比较两个值。你总是在关系运算符的两边写上变量、字面量或表达式——或者任意二者的组合。*

在深入研究if语句之前，来看一些关系运算符以及它们的真正意思。普通运算符产生数字的结果。关系表达式产生真（true）或假（false）的结果。比较两个

趣闻轶事
为了区分=和==的不同，请记住要用两个等号来双重检查某物是否相等。

数据值时，结果可能为真，也可能为假。例如，给定下面的数值：

```
int i = 5;
int j = 10;
int k = 15;
int l = 5;
```

下面的语句的结果是真：

```
i == l
j < k
k > i
j != k
```

下面的语句的结果是假：

```
i > j
k < j
j == l
```



警告 只有相似的数值应该出现在关系运算符的两边。换句话说，不要比较字符和浮点数。如果你不得不比较两个不相似的数值，请用强制类型转换得到一致的数据类型。



警告 每当C语言对关系运算符进行求值时，总是产生数值1或0。为真时结果是1，为假时结果是0。下面的语句把1赋给变量a，把0赋给变量b:

```
a = (4 < 10);  /* (4 < 10) is true, so a 1 is put in a */

b = (8 == 9);  /* (8 == 9) is false, so a 0 is put in b */
```

在程序中会经常用到关系运算符，因为你经常会想知道销售额（存储在变量中）是否超过了预定的目标，工资表的计算是否一致，某产品是有存货还是需要定购。现在你只学习了关系运算符的基础知识。下一节会解释如何使用它们。

11.2 使用 if 语句

if语句用关系运算符来执行数据测试。下面是if语句的格式：

```
if (condition)
{ block of one or more C statements; }
```

*condition*外面的括号是必需的。*condition*是像前一节中提到的那些关系测

试。*block of one or more C statements*称为if语句的主体。如果if语句的主体包含了多条语句，那么*block of one or more C statements*外面的大括号是必需的。



线索　尽管如果if只包含了一条语句，就可以不需要大括号，但是最好总是写上大括号。如果你后来又增加语句到if的主体中，就不用再去加大括号了。如果大括号包含了多条语句，那么大括号里的语句就叫做复合语句(compound statement)。

下面是一条if语句：

```
if (age < 18)
  { printf("You cannot vote yet\n");
    yrs = 18 - age;  /* Calculates how many years
                        until the user can vote */
    printf("You can vote in %d years.\n", yrs); }
```

对于程序员来说，这条if语句这样理解："如果名为age的变量包含一个小于18值，就打印消息并计算yrs的值。否则，就不进行打印和计算。无论发生什么，一旦if完成了它的工作，程序都将从跟在if的主体之后的语句继续执行下去。"



说明　附录B中21点游戏程序的main()函数询问玩家是否要领取另一张牌。这里用了一条if语句来判断用户想要做什么。

11.3　否则……：用 else 语句

在前一节中，你看到了如何编写当关系测试为真时的if语句，且如果关系测试是假，就什么都不做。本节要解释else语句，它可以加到if语句中。使用else语句，你可以准确地指定当关系测试为假时可以做些什么。下面是if-else组合语句的格式：

```
if (condition)
  { block of one or more C statements; }
else
  { block of one or more C statements; }
```

下面是if-else语句的例子：

```
if (age < 18)
  { printf("You cannot vote yet\n");
    yrs = 18 - age;
    /* Prints an appropriate message,
       depending on user's age */
```

```
    printf("You can vote in %d years.\n", yrs); }
else
  { printf("You've made it to adulthood.\n"); }
```

如果age小于18，将执行if语句的主体。如果age不小于18，将执行else语句的主体。



线索 只在if或else语句的主体中的可执行语句末尾加上分号。不要在if或else语句后面加分号。分号只能出现在完整的语句末尾。



请跳过这里，这有点儿难 与if语句的主体一样，如果else语句的主体只包含一条语句就可以不用写大括号。但是最好在任何时候都写上大括号。

下面的程序在用户工资超过预置的免税线时，计算收入税：

```
/* Computes simple tax */
#include <stdio.h>
/* Defined constant holds salary level trigger */
#define MAXSAL 18000.00
main()
{
  float salary, taxAmt;
  float taxRate = .35;
  printf("How much did you earn last year? ");
  scanf(" %f", &salary);
  /* No tax for small salaries */
  if (salary >= MAXSAL)
    { taxAmt = salary * taxRate; }
  else
    { taxAmt = 0.0; }  /* None for low salary */
  printf("Your tax is %.2f\n", taxAmt);
  return 0;
}
```

下面是程序的两次运行，表明了根据用户薪水的不同产生输出也不同：

```
How much did you earn last year? 24390.40
Your tax is 8536.64

How much did you earn last year? 16558.21
Your tax is 0.00
```



奖励

- 用关系运算符来比较数据。

- 请记住真的关系结果产生1，而假的关系结果产生0。
- 用if语句来比较数据，用else语句来指定当if语句测试失败时做些什么。
- 在if和else语句的代码主体部分加上大括号。大括号中的代码是否被执行，取决于关系比较的结果。



陷阱

- 不要在关系运算符两边用不同数据类型的值。
- 不要在if或else语句后面加上分号。分号只能出现在if或else语句的主体中每条语句的末尾。

11.4　小结

本章展示了测试数据并根据测试的结果选择执行某一段代码还是另一段代码的方法。运行程序时，你不会总是想执行同样的代码，因为数据不会总是一样的。电脑必须能够查看并处理数据（这是为什么叫做数据处理的原因）。

关系运算符和if语句一起可以产生数据的真假值分析。

1. 代码范例

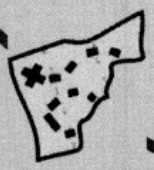
```
printf("How much did you make last year? ");
scanf(" %f", &salary);
if (salary > 100000.0)
  { printf("Wow! You really worked hard despite ");
    printf("what some might think!"); }
else
  {  sal100 = 100000.0 - salary;
     printf("Only $%.0f more to reach the century mark!",
            sal100); }
```

2. 代码分析

用户输入一个薪水值时，程序会根据它与$100 000的关系做决定。如果用户挣的超过$100 000（我们都应该这么能干），就会打印出一条祝贺的消息。如果用户挣的少于$100 000，则会打印出另一条消息，告诉用户他与$100 000相比还差多少（我们中的大多数同志仍需继续努力）。

else语句确保在if语句之后只执行一段代码。

12



写一个存货清单程序
101



21点游戏程序
103



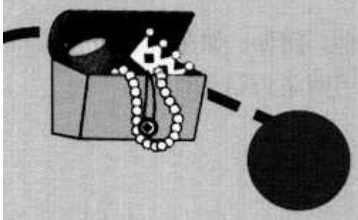
编写清楚的代码
101



术语逻辑和
关系
98



不要过多使用！
104



第 12 章

逻辑运算符

使用逻辑运算符

本章内容

获取逻辑
98

逻辑运算符的优先级
103

有时候，第11章中描述的关系运算符不能表达所有的测试条件。例如，如果想测试一个数值的或字符的变量是否在某一特定范围内，就不得不用两条if语句，像这样：

```
if (age >= 21)    /* See if 21 <= age <= 65 */
{ if (age <= 65)
    {
     printf("The age falls between 21 and 65.\n");
    }
}
```

尽管使用嵌套的if语句没有任何错误，但是这样写不太直观，并且逻辑变得有点复杂。通过使用本章将介绍的逻辑运算符（logical operator），你可以在一条if语句中将多个关系测试组合在一起，这样可以使程序更清晰。



说明　不要让术语逻辑（logical）和关系（relational）使你觉得这两组运算符很难。只要你理解了每个运算符是如何工作的，你就不必在意它们作为一组运算符时的称呼。



线索　关系运算符只是测试两个值的关系（把它们相互比较）。逻辑运算符把关系运算符组合起来。

12.1　获取逻辑

有3种逻辑运算符（参见表12-1）。有时候逻辑运算符又称为复合关系运算符（compound relation operator），因为它们能够把多个关系运算符组合起来（参见前面的线索部分）。

表12-1　逻辑运算符

逻辑运算符	意　义
&&	与
\|\|	或
!	非

逻辑运算符出现在两个或更多的关系测试之间。例如，下面是用了逻辑运算符

的3个if语句的第一部分：

```
if ((age >= 21) && (age <= 65)) {
```

和

```
if ((hrsWorked > 40) || (sales > 25000.00)) {
```

和

```
if (!(isCharterMember)) {
```



线索 如果用逻辑运算符把两个关系运算符组合起来或者用！（非）运算符对一个关系取反，那么在if语句后面的整个表达式需要加上括号。下面的写法是不允许的：

```
if !isCharterMember {     /* Not allowed */
```

当然，前述的if语句还远不止这些内容，但现在为了保持问题简单，if语句的主体就不写出来了。

逻辑运算符的含义与普通英语类似。例如，看一看与下面的代码行相应的口头表述：

```
if ((age >= 21) && (age <= 65)) {
```

用口语来说就是：

```
"If the age is at least 21 and no more than 65,..."
if ((hrsWorked > 40) || (sales > 25000.00)) {
```

用口语来说就是：

```
"If the hours worked are more than 40 or the sales are more than
$25000,..."
if (!(isCharterMember)) {
```

用口语来说就是：

```
"If you aren't a charter member, you must..."
```

很容易你就能明白，这3句话准确地描述了由3条if语句完成的测试。你经常会在两个条件之间加上与（and），比如“如果你把垃圾拿出去并（and）打扫房间，你就可以玩了。”图12-1和图12-2展示了与和或的不同。



图12-1　与条件要求两边都为真



图12-2　或条件要求有一边为真



说明 重新读一下可能会对孩子说的那句严厉的话。与条件给出一个严格的要求，只有两个工作都完成了结果才能发生。这也是&&要做的。为了执行if语句的主体，&&两边的条件必须都为真。

继续讨论||（或）运算符。你可能对小孩仁慈的话，会说："如果你把垃圾拿出去或（or）打扫房间，你就可以玩了。"或不是限制性的。只需||一边的条件为真（两边的条件也可以都为真）。如果有一边条件为真，结果就会发生。||运算符也是这样的。为了执行if的主体，只需||一边的条件为真。

!（非）运算符将真或假的条件取反。真变为假，假变为真。这听起来使人糊涂，确实如此！尽量少用!运算符。你总是可以通过将逻辑取反重写逻辑表达式以避免使用!。例如，下面的if语句：

```
if ( !(sales < 3000)) {
```

与下面的if语句完全相同：

```
if ( sales >= 3000) {
```

如你所见，你可以通过移除!以及使用相反的关系运算符来移除!，并把否定的语句变为肯定的测试。



线索 假设你想编写一个存货清单的程序，它测试了某个货物的数量是否减少为0。if语句的第一部分可能像这样：

```
if (count == 0) {
```

因为只有当count为0时if才为真，你可以像下面这样重写语句：

```
if (!count) {  /* Executes if's body only if count is 0 */
```



警告 重说一遍，!会使程序变得有点混乱。即使用了!可能会减少一点输入，但是清楚的代码比需要技巧的代码好，而且用if(count == 0)可能会更好，尽管用了!或许会节省几微秒的时间。

通过使用&&运算符，如果用户的姓以字母P到S开头，下面的程序就打印一条消息，而如果以其他字母开头，就打印另一条消息。

```
#include <stdio.h>
main()
{
   char name[25];
   printf("What is your last name? ");
   scanf(" %s", name);

   if ((name[0] >= 'P') && (name[0] <= 'S'))
      { printf("You must go to room 2432 ");
        printf("for your tickets.\n"); }
   else
      { printf("You can get your tickets here.\n"); }
   return 0;  /* Leaves the program to DOS or C editor */
}
```



说明　如果把&&改成||，程序会有什么不同？会出现第一条消息还是第二条消息？答案是第一条。每个人都会被送到2432房间。从A到Z的任何字母或者大于P或者小于S。上面程序中的测试必须用&&，因为只有那些名字在P到S之间的人才能用房间2432。

下面的代码段要求用户输入答案Y或N。代码中用||来确保用户输入了一个正确的值。

```
printf("Is your printer on (Y/N) ?");  /* Request Y or N */
scanf(" %c", &ans);
if ((ans == 'Y') || (ans == 'N'))
  { /* Gets here if user typed a correct answer */
    if (ans == 'N')
      { printf("** Turn the printer on now. ***\n");
      }
  }
else
  {
    printf("You did not enter a Y or N.\n");
    exit(1);
  }
```



线索　你可以用逻辑运算符把两个以上的关系运算符组合起来，但是在一个语句中写太多关系运算符会引起混乱。下面的写法就有点太多了：

```
if ((a < 6) || (c >= 3) && (r != 9) || (p <= 1)) {
```

试着让组合后的关系测试简单一些，这样程序不仅易读而且易于维护。

12.2 逻辑运算符的优先级

因为逻辑运算符出现在运算符优先级表中，所以它们像其他运算符一样具有优先级。研究运算符的优先级会发现&&运算符优先于||。因此，下面的逻辑：

```
if (age < 20 || sales < 1200 && hrsWorked > 15) {
```

被C语言理解成下面这样：

```
if ((age < 20) || ((sales < 1200) && (hrsWorked > 15))) {
```

要尽量使用括号。括号有助于分清运算符的优先级。如果不用括号，C语言也不会混乱，因为它清楚地知道运算符的优先级。但是，读程序的人不得不琢磨哪个先做哪个后做，并且括号有助于对运算符进行分组。



请跳过这里，这有点儿难 附录B中的21点游戏程序在findWinner()函数中使用&&和||来判断胜利的一方。尽管程序的很多内容你还没学过，但是你应该能够读懂程序的逻辑并且大体了解程序在做什么。

假设有一个教师想奖励她的学生，这些学生要表现很好并且很少缺课。另外，这些学生还要加入了3个学校社团或者参加了2项体育活动。唷！你一定承认，不仅奖励是应得的，而且要整理出符合这些要求的所有学生也是很难的。

在C程序代码中，如果某学生达到了老师预设的奖励标准，下面的if语句测试就为真：

```
if (grade > 93 && classMissed <= 3 && numActs >= 3 || sports >=
2) {
```

需要解释的有很多。这条语句不仅难以理解，而且它有一点错误。||比较最后进行（因为||的优先级比&&低），但是||比较本来应该发生在第二个&&之前。如果你也糊涂了，这很正常！长的组合关系测试通常会这样。下面是用口语对上面的if语句操作的描述：

“如果学生的成绩高于93并且他的缺课数小于等于3并且社团活动大于等于3，或者如果他参加了两项以上的体育活动……”

问题是学生只需要参加体育活动就可以得奖。最后的两个关系（用||分隔）必须在第三个&&之前比较。口头描述应该是这样：

“如果学生的成绩高于93并且他的缺课数小于等于3并且或者社团活动大于等于3或者他参加了两项以上的体育活动……”

下面的if，加上了正确的括号，不仅使if语句更准确，也使其意义更清晰：

```
if ((grade > 93) && (classMissed <= 3) && ((numActs >= 3) ||
(sports >= 2)) {
```

如果你愿意，也可以像下面这样把这种长的if语句分成两行以上：

```
if ((grade > 93) && (classMissed <= 3) &&
   ((numActs >= 3) || (sports >= 2))) {
```

一些C程序员甚至觉得两条if语句比4个关系测试更清晰，比如下面的语句：

```
if ((grade > 93) && (classMissed <= 3)
   { if ((numActs >= 3) || (sports >= 2))
       { /* Reward the student */ }
```

你最后选择的样式主要取决于你最喜欢的是什么，你觉得用什么最舒服，对你来说也就最易于维护了。



奖励

- 用逻辑运算符把关系运算符连接起来。
- 当运算符两边都为真时，整个条件才为真，这种情况下要用&&。
- 当运算符的一边（或两边都）为真时，整个条件就为真，这种情况下要用||。



陷阱

- 不要过多使用!。大多数否定的逻辑可以取反（<变成>=，>变成<=）以避免使用非运算符。
- 不要把太多的关系运算符组合在一个表达式中。

12.3　小结

本章讲解了逻辑运算符。尽管关系运算符可以用来测试数据，不过逻辑运算符&&和||能够把多个关系测试组合在一个语句中并据此执行代码。

通过本章可以看出，我们经常在现实生活中使用类逻辑运算符。我们经常要根据几种条件做决定。

1. 代码范例

```
printf("What is your grade average for last semester? ");
scanf(" %f", &avg);
printf("How many hours did you take? ");
scanf(" %d", &hours);
if ((avg > 90.0) && (hours > 15))
   { printf("You are a top-notch student!\n");
     printf("Keep up the good work!"); }
```

2. 代码分析

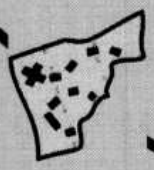
这段代码是一个简单的例子，它是另一位教师鼓励好学生的程序。程序测试了两个关系：学生的平均分和学习的课时。只靠平均分并不足以对学生进行鼓励，因为他可能只上了一门课。同理，仅仅因为上了很多课也不意味他很出色。

因此，最好是把平均分和课时数都组合到一个用了&&的复合关系测试中。如果学生的平均分很高并且上了很多课，学生就会收到继续努力的鼓励。

这个例子比本书前一节的代码更简单，因为我们最好不要把3个以上的关系运算符组合在一起，而应该使if语句测试尽量简单。



13



使用括号对条件运算符的不同部分进行分组
108



只有变量才能自增或自减
110



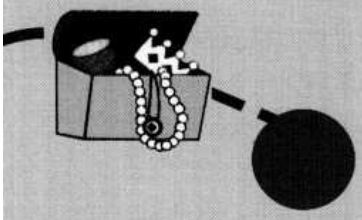
把赋值提到条件运算符外面
109



条件运算符可以减少输入
109



前缀和后缀的位置
112



第 13 章

更高级的运算符

其他运算符

本章内容

条件运算符
108

运算符++和--
110

运算符sizeof()
111

请多一点耐性！你已经学习了几乎所有的C语言运算符。除了将在第24章中介绍的一些更高级的运算符，本章将把运算符优先级表介绍完。下面将讲解条件运算符（condition operator）、递增运算符（increment operator）和递减运算符（decrement operator）。

C语言的运算符有时候替换了其他编程语言中更冗长的命令。各种各样的运算符不仅加快了程序的开发时间，它们也比命令编译得更高效而且运行得更快。C语言的运算符在使其成为高效的语言方面做了很多贡献。

13.1　条件运算符

条件运算符是唯一需要3个参数的C运算符。除法、乘法和其他大多数运算符都只需要两个参数，而条件运算符需要3个参数。尽管条件运算符的格式看起来有点复杂，但是你会发现它不仅使一些逻辑简单化了而且使用起来也更直观。

条件运算符的形式为?:。下面是它的语法格式：

```
relation ? trueStatement : falseStatement;
```

趣闻轶事
有些人把?:叫做三元运算符，因为它有3个参数。

*relation*可以是任何关系测试，如age >= 21或sales <=25000.0。你还可以用第12章中学过的逻辑运算符把多个关系运算符组合起来。*trueStatement*可以是任何合法的C语句，*falseStatement*也一样。下面是条件运算符的一个例子：

```
(total <= 3850.0) ? (total *= 1.10): (total *= 1.05);
```



线索　括号不是必需的，但它们确实有助于对条件运算符的3个部分进行分组，这样能够更容易看清。

如果第一组括号里的测试结果为真，那么就执行*trueStatement*。如果第一组括号里的测试结果为假，就执行*falseStatement*。上面的条件运算符与下面的if-else语句的功能完全相同：

```
if (total <= 3850.0)
  { total *= 1.10; }
else
  { total *= 1.05; }
```

这句话使C语言根据关系测试的结果，把total乘上1.10或1.05。



说明 几乎任何if-else语句都可以写成条件语句。条件语句需要更少的输入，你也不会漏掉大括号，并且条件语句运行也更高效，因为它会被编译成更简练的代码。



线索 当你把条件运算符看成“问号（？）提出了一个问题”时，它的格式就很明白了。牢记这一点，你可以把前面的例子陈述如下：total<=3850.0是否成立？如果成立，就做第一件事情；否则，就做第二件事情。



警告 C程序员不喜欢前面的条件运算符中的冗余。如你所见，变量total出现了两次。两次都被赋了一个值。遇到这种情况时，请把赋值提到条件运算符外面来：

```
total *= (total <= 3850.0) ? (1.10): (1.05);
```

不要把每一个if-else语句用条件运算符替换。很多时候if-else语句更易读，而且有些条件语句太复杂，不容易压缩成条件运算符。但是，对于简单的if-else语句，条件运算符提供了另外一个很好的方法。

条件运算符比if语句还有一个优点：条件运算符可以出现在if语句不能出现的地方。下面的printf()中，如果梨的个数大于1，就打印出最后的s：

```
printf("I ate %d pear%s", numPear, (numPear>1) ? ("s.") : (".));
```

如果numPear的值大于1，你将会看到：

```
I ate 4 pears.
```

但如果只有一个梨，你会看到：

```
I ate 1 pear.
```



说明 你可能想知道为什么条件运算符是?:，但是问号和冒号绝不会出现在相邻的位置。就是这样。如果说成条件运算符像是中间包含某些内容的一个问号和一个冒号，这样也太繁琐了。

13.2　运算符++和--

条件运算符操作3个参数，而*递增*（increment）和*递减*（decrement）运算符只操作一个参数。递增运算符给变量加1，递减运算符给变量减1。

如果你在对某些项进行计数（比如昨天来商店购物的顾客数）或者逆向计数（比如当有人买走货物时，从存货清单中移除货物数目）时，就需要对变量递增和递减操作。在第10章中，你学过如何用复合运算符递增和递减变量。下面，你将学到两种运算符，它们能够更容易地实现相同的功能。递增运算符是++，递减运算符是--。给变量count加1时，可以这样写：

```
count++;
```

你也可以这样写：

```
++count;
```

递减运算符做类似的事情，只是从变量中减去1。可以这样写：

```
count--;
```

也可以这样写：

```
--count;
```

如你所见，这两个运算符可以出现在变量的任何一边。如果运算符出现在左边，称为*前缀递增*（prefix increment）或*前缀递减*（prefix decrement）运算符。如果运算符出现在右边，则称为*后缀递增*（prefix increment）或*后缀递减*（prefix decrement）运算符。



请跳过这里，这有点儿难　不要把递增或递减运算符用于字面量或表达式。它们只能用于变量。你不可能看到：

```
---14;     /* Don't do this!  */
```

前缀运算符和后缀运算符独立使用时，会产生相同的结果。只有当它们与其他表达式组合在一起时，才会出现一点“陷阱”。看一看下面的代码：

```
int i = 2, j = 5, n;
n = ++i * j;
```

问题是：当语句执行完时n是多少？j的值很容易看出来，因为j没有改变且仍

然是5。++确保i会递增，于是i变成3。关键是判断i在什么时候递增。如果i在做乘法之前递增，n就是15，但如果i在做乘法之后再递增，n就是10。

答案取决于前缀（prefix）和后缀（postfix）。如果++或--是前缀，C语言在本行的任何事之前执行它。如果++或--是后缀，C语言在本行的每一件事之后执行它。因为前面代码中的++是前缀，在做乘法之前i就递增到3。下面的语句在i乘以j之后使i递增，并把结果存储在n中：

```
n = i++ * j;  /* Puts 10 in n and 3 in i */
```

能够在使用变量的同一个表达式中递增变量意味着减少了程序的工作量。前面的语句代替了下面的两条（在其他编程语言中不得不写的语句）：

```
n = i * j  /*No comment*/
i = i + 1
```



警告 ++和--运算符非常高效。如果你关心这些事（大多数人都不关心），++和--编译后只生成一条机器语句，而用+1或-1进行加1或减1不会总是这么高效。

13.3 运算符 sizeof()

趣闻轶事
sizeof() 看起来根本不像一个运算符。它看起来（且在功能上）像是一个函数调用。

可以用sizeof()查明存储任何类型的数据所占用的内存单元。尽管大多数C编译器用2个字节来存储整数，但并非所有的编译器都是这样。要查明整数和浮点数到底占用多少内存，可以用sizeof()。下面的语句正是做了这件事：

```
i = sizeof(int);    /* Puts size of integers into i */
f = sizeof(float);  /* Puts size of floats into f */
```

sizeof()可以对变量和数据类型都起作用。如果需要知道变量和数组占用了多少内存，也可以对它们应用sizeof()。下面的代码段展示了如何使用它：

```
char name[] = "George Paul";
int i = 7;
printf("The size of i is %d\n", sizeof(i));
printf("The size of name is %d", sizeof(name));
```

下面是一种可能的输出：

```
The size of i is 2
The size of name is 12
```

取决于你的电脑和C编译器，由于整数大小的差别，输出可能会不同。请注意上面的字符数组的大小是12，其包括了null零。



线索　字符串的长度和字符串的大小是两个不同的值。长度是指不包含null零的最大字节数，可以通过`strlen()`得到。字符串的大小是指存放字符串所占用的字符数，包含了null零。



请跳过这里，这有点儿难　尽管现在看起来`sizeof()`好像没有什么用，随着你进一步学习C语言你将会发现它很有用。



奖励

- 用条件运算符代替简单的`if-else`语句以提高效率。
- 条件运算符需要3个参数。额外的括号通过把3个参数彼此分开有助于区分它们。
- 使用`++`和`--`来递增和递减变量，而不是使用赋值语句加1和减1或者用`+=`和`-=`。`++`和`--`十分高效。



陷阱

- 不要在条件运算符`?:`的每一边都重复赋值语句。把变量和赋值运算符放到条件运算符的左边以提高效率。
- 不要认为前缀和后缀总是产生相同的值。只有当应用于一个变量时，前缀和后缀才是相同的。如果把`++`和`--`与其他变量和表达式组合起来，前缀和后缀的位置对于得到想要的结果来说至关重要。

13.4　小结

本章扩展了关于C语言运算符的知识，并且展示了大多数未学的运算符。讲解了最长的C运算符`?:`和最短的C运算符`--`和`++`。理解这些运算符不用花很大力气，但这些运算符却很强大而且无需像其他编程语言那样编写完整的语句。最后，还有一种C运算符看起来根本不像运算符。`sizeof()`运算符返回其括号里的任何对象所占用的内存单元数目。

1. 代码范例

```
/* Works with an age and integer */
#include <stdio.h>
main()
{
  int age, intSize;
  float gift;
  printf("\nHow old are you? ");
  scanf(" %d", &age);
  if (age < 18)
    { gift = 5.00; }
  else
    { gift = 10.00; }
  gift = (age < 18) ? 5.00 : 10.00;  /* Does the same
                                        as the if */
  printf("Your gift is $%.2f\n", gift);
  age++;  /* Adds 1 to age */
  printf("In a year, you'll be %d years old.\n", age);
  intSize = sizeof(int);
  printf("Integers take %d memory locations.", intSize);
  return 0;
}
```

2. 代码分析

if语句是多余的。如果去掉if语句，程序的执行会与有if语句的程序完全一样。条件运算符与if语句做的是相同的事情。条件运算符根据用户的年龄，分配5美元或10美元的礼物。

然后程序给用户的年龄加上1，这样用户就能看到他们一年后的年龄（好像他们自己不会算一样）。最后，代码打印出一条消息，告诉用户在运行程序的机器上整数占用的内存数量。



Part 3

第三部分

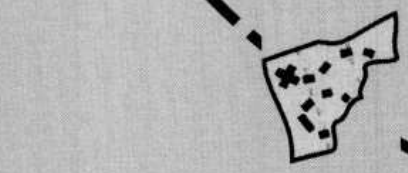
保持控制

本部分内容

- 第 14 章　循环
- 第 15 章　其他循环
- 第 16 章　终止循环
- 第 17 章　测试多个值
- 第 18 章　输入和输出
- 第 19 章　更多有关字符串的内容
- 第 20 章　更高级的数学函数



14



while语句主体
118



do和while像循环主体
的包装物一样
121



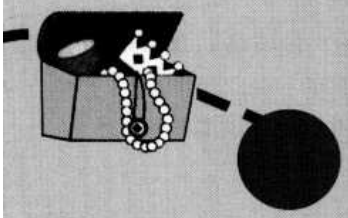
while和if是两
条独立的语句
119



如果条件为假
119



无限循环
121



第 14 章

循　　环

使用while和do-while循环

本章内容

while循环
118

使用while语句
119

使用do-while语句
120

既然你已经学习了运算符，你就做好了用程序玩“环路火车”的准备。循环（loop）是指一段重复执行多次的代码。你绝对不希望循环永远重复下去。这叫做无限循环（infinite loop）。完成了设置的任务后，你写的循环应该结束（如果写得正确，当然会的）。

为什么要让程序循环？想一想电脑可以用来做一些人们不愿意做的事情，就会明白了。电脑从来不会感到厌烦，于是最好给它们分配一些普通的重复的任务，而把需要思考的任务留给人来做。你一定不愿意花钱请人把成百上千的工资表图表相加，而且很少有人愿意做这件事。电脑程序可以做这种重复性的工作。当电脑循环完成所有图表的计算后，人们可以对其结果进行分析。

如果想把一些图表相加，打印公司过去12个月的销售额，或者统计参加电脑培训班的学生数，就需要用到循环。本章将介绍两种使用while命令的常见C语言循环。

14.1　while 循环

while语句总是出现在循环的开始或结尾。使用while的最简单的循环叫做while循环。另一种叫做do-while循环。很快你就会看到。下面是while的语法格式：

```
while (condition)
{ block of one or more C statements; }
```

*condition*是一个关系测试，它与if语句中的关系测试*condition*一样。*block of one or more C statements*叫做while的主体。



线索　只要*condition*为真，while语句的主体就会一直重复执行。回想一下if是如何工作的：如果*condition*为真，if语句的主体就执行。if语句的主体只会执行一次，然而，while的主体可以执行很多次。

图14-1帮助说明了if语句和while语句的共同点和不同点。这两种命令的格式有一个共同点，就是当while的主体包含多条语句时，需要加上大括号。即使while的主体只包含一条语句，也应该加上大括号，这样当你将来添加语句时，就不用再加大括号了。千万不要在while的括号后面加上分号。分号只出现在while的主体中的语句后面。

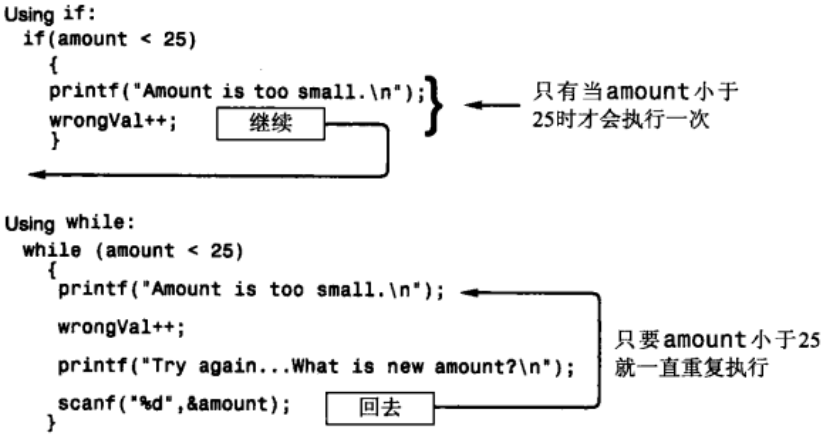


图14-1 if语句的主体只执行一次，而while语句的主体可以重复执行多次



警告 图14-1中的两条语句很相似，但它们做的是不同的事。while和if是做两件独立事情的两条独立的语句。

你必须以某种方式修改while循环*condition*中的变量。如果不这样做，while将会永远循环下去，因为每次循环时都测试相同的*condition*。因此，为了避免无限循环，你要确保while循环的主体会修改*condition*中的某个变量，这样最终*condition*会变成假并且程序会继续执行while循环后面的语句。



说明 和if语句一样，while语句也可能永远不执行！如果第一次进入while语句时，*condition*为假，那么while的主体就不会执行。

14.2 使用 while 语句

如果想重复执行一段代码直到某一特定条件变为假，使用while是一种办法。附录B中的21点游戏程序在dispTitle()函数中灵活地使用while进行清屏。下面把函数重复一遍，以供你复习：

```
/* Clears everything off the screen */
void dispTitle(void)
{
  int i = 0;
```

```
  while (i < 25)    /* Clears screen by printing 25 blank */
  { printf("\n");   /* lines to "push off" stuff that     */
    i++; }          /* might be left over on the screen   */
                    /* before this program                */
  printf("\n\n*Step right up to the Blackjack tables*\n\n");
  return;
}
```



警告　几乎每一种C编译器都带有内置的清屏函数。问题在于这些清屏函数都不与ANSI C标准相兼容。因此，我用了另一种办法来实现在每一次21点游戏程序之前擦除屏幕。while循环打印出25个新行。当光标到达屏幕底端时，屏幕就向上滚动，并且屏幕上的内容被从上方移走。这个简单而且普通的清屏函数是用while写的，与所有的ANSI C编译器（比如你用的那个）相兼容。



说明　用循环的方法来实现清屏不是最好的，但却是最通常的方法。如果想从编译器的手册中找到清屏函数来替换这个while循环，完全可以。

回到用来清屏的实际的while代码：变量i的初值设为0。当while第一次执行时，i小于25，while语句中*condition*为真并且执行while的主体。在主体中，一个新行被发送到屏幕上，并且i的值增1。第二次测试*condition*时，i的值为1，但1还是小于25，于是主体再次执行。主体连续执行直到i的值增加到25。因为25不会小于25（它们是相等的），*condition*变为假并且循环停止重复。然后就可以执行程序的其他部分了。



线索　如果在这个清屏while中，i的值没有增加，那么将会永远执行printf()或者直到按下Ctrl+Break才能终止程序。

14.3　使用 do-while 语句

while也可以与do语句联合起来使用。联合使用时，通常称为do-while语句或do-while循环。do-while与while循环几乎完全一样。下面是do-while的语法格式：

```
do
  { block of one or more C statements; }
while (condition)
```



请跳过这里，这有点儿难 do和while就像循环主体的包装物一样。另外，如果主体包含多条语句，就需要加上大括号。

只有当循环的主体必须至少执行一次时，才用do-while循环而不是while循环。*condition*位于do-while循环的底部，因此直到第一次循环结束后C语言才能测试*condition*部分。

21点游戏程序要求用户键入H或S——换句话说，hit或stand，意思是说是否领取另一张牌。然而，用户经常会输入无关的内容。因此，当希望用户提供固定的几种输入时，如H或S，你应该检查用户输入的内容以确保它是你想要的一个答案。下面是do-while的例子，它与21点游戏程序中用的循环类似，始终要求用户输入答案直到输入正确为止：

```
do {
  ans = getAns("Hit or stand (H/S)? ");
} while ((ans != 'S') && (ans != 'H'));  /* Loop if bad
                                            answer */
```

奇怪的getAns()函数调用只是实现了一个功能，即显示Hit or stand (H/S)?消息并且从用户获取一个字符存放到ans中。现在，研究一下do-while的形式，因为这是你所要关心的核心内容。



线索 第19章介绍了如何用内置的toupper()函数测试大写的Y或N，或者小写的y或n。



奖励

- 需要重复执行一段代码时，使用while或do-while循环。
- 确保while或do-while循环的主体修改了*condition*中的某个变量，否则循环将永远重复下去。
- 请记住循环与if语句不同，因为if语句的主体只执行一次，而如果*condition*为真，循环会执行多次。



陷阱

- 不要在while的*condition*外的括号后面加上分号。如果加了分号，将会导致无限循环。

- 不要认为用户会按照你期望的方式回答问题。始终要求输入答案直到用户输入了你想要的内容。

14.4　小结

本章展示了如何重复执行一段代码。while和do-while循环都重复执行主体中的语句。while和do-while循环大括号中的所有代码都不止执行一次。当给定的关系条件为真时，while和do-while语句会重复执行它们的代码段。

二者的区别在于用来控制循环的关系测试所处的位置不同。while语句在循环的顶部测试，而do-while语句在循环的底部测试，这也致使它的所有语句至少执行一次。

1. 代码范例

```
/* Prints 1 to 20 twice */
#include <stdio.h>
main()
{
  int ctr = 1;
  while (ctr <= 20)
    { printf("%d \n", ctr);
      ctr++; }
  ctr = 1;
  do
    { printf("%d \n", ctr);
      ctr++; }
  while (ctr <= 20);
  return 0;
}
```

2. 代码分析

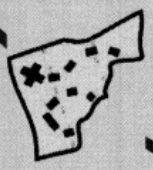
上面的代码展示了本章介绍的两种循环。程序打印出1到20的数字两遍。程序的前半部分用while语句来控制循环，后半部分用do-while语句来控制循环。如你所见，两种循环可以做相同的事情。选择用while还是do-while循环通常不太重要。只有当第一遍循环很重要并且你想让循环至少执行一次（用do-while）时，你会选择其中一个，或者也可能根本不是这么回事。

顺便提一下，如果printf()像这样包含了ctr的递增：

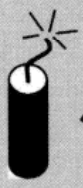
```
{ printf("%d \n", ctr++); }
```

你就不需要单独的递增语句了。然而，单独的递增语句会使程序更易读，而在大多数情况下，可读性比效率高更重要。

×15



提高循环的可读性
126



do-while循环测试
128



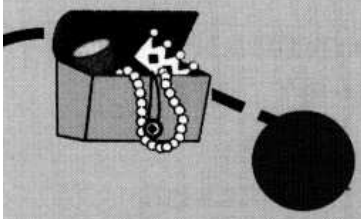
for循环执行的顺序
127



模拟for循环
126



不要忘记for循环中的分号
129



第15章

其他循环

for循环

本章内容

为了重复
126

使用for循环
128

还有另外一种C循环叫做for循环。for循环比while和do-while循环提供了更多的控制。使用for循环，可以准确指定循环的次数，而使用while循环，只要条件为真就必须继续循环下去。

在C程序中3种循环都有各自发挥作用的机会。很多时候，某一种循环比其他循环会更适合程序的需求。例如，当处理顾客从存货中买走物品时的顾客定单时，就需要用while循环。也就是说，每当顾客走出商店时，程序就处理定单。如果正好有100个顾客买了东西，while循环将会执行100次。晚上，你可能要把当天的顾客购买数量加在一起看看总数是多少。这时你就可以用for循环，因为你准确地知道要循环多少次。



说明　通过递增计数器变量，可以用while循环模拟for循环。你也可以用for循环来模拟while循环！因此，使用哪种循环最终取决于应用的场景。

15.1　为了重复

从本节的标题可以看出，for循环对于控制代码段的重复执行很重要。for循环的语法格式有一点奇怪：

```
for (startExpression; testExpression; countExpression)
{ block of one or more C statements; }
```

可能用实际代码写的简短例子更容易理解：

```
for (ctr = 1; ctr <= 10; ctr++)
  { printf("Still counting... ");
    printf("%d\n", ctr);
}
```

下面是这条for语句工作的方式：当for循环开始执行时，*startExpression*即ctr = 1;被执行。在任何for循环中，*startExpression*都只被执行一次。然后对*testExpression*进行测试。在本例中，*testExpression*是ctr <= 10;。如果它为真——这段代码中第一次测试会为真——for循环的主体就被执行。当for循环的主体执行结束时，*countExpression*被执行（ctr加1）。



线索　如你所见，把for循环的主体缩进有助于将其与程序的其他部分区分开来，提高循环的可读性。对于其他类型的循环也是如此，比如do-while循环。



警告 不要想一下子就搞明白了——甚至一段当中。让我们把它变得简单。图15-1中的线条，说明了for循环执行的顺序。按照这条线走时，请重新阅读上面的段落，你会更容易理解。

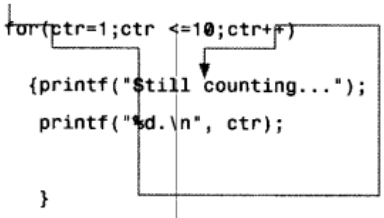


图15-1 for循环的顺序



说明 因为需要嵌入的分号，所以for循环的语法格式看着有点奇怪。分号确实只能出现在可执行语句的末端，不过for循环中的语句也是可执行的。例如，初始化表达式ctr = 1;，在循环开始之前执行完成，如图15-1所示。

下面是写成while语句的功能相同的循环：

```
ctr = 1;
while (ctr <= 10)
  { printf("Still counting... ");
    printf("%d\n", ctr);
    ctr++;
}
```

下面是输出：

```
Still counting...1
Still counting...2
Still counting...3
Still counting...4
Still counting...5
Still counting...6
Still counting...7
Still counting...8
Still counting...9
Still counting...10
```



线索 如果按照图15-1中的指导线走并阅读前面讲解的while循环，你会明白for循环和while循环怎样做相同的事情。在while循环前面的ctr = 1;是for循环中执行的第一条语句。



请跳过这里，这有点儿难　do-while循环不能真正表示for循环，因为在for循环中关系测试发生在主体之前，而在do-while循环中关系测试发生在主体之后。do-while循环的测试总是处于循环的底部。

15.2　使用 for 循环

趣闻轶事
如果用BASIC或QBasic编程，每个for语句就需要一个next。在C语言中，并不需要。

for循环读起来与日常生活中说话的方式很像。看一看下面的语句：

分别计算出45个雇员的工资并打印支票。

这句话没有任何含糊之处。有45个雇员、45份工资要计算和45张支票要打印。for循环并不总是像前面的例子那样让ctr的值增加。下面的for循环，在打印消息之前让计数器的值减少：

```
for (cDown = 10; cDown > 0; cDown--)
  { printf("%d\n", cDown); }
printf("Blast off!\n");
```

下面是代码的输出：

```
10
9
8
7
6
5
4
3
2
1
Blast off!
```



线索　如果for括号中的最后一个表达式以某种方式递减，那么初始值必须比测试的值大，这样循环才会执行。在前面的for语句中，初始值10比*testExpression*的0大。

下面的for循环的计数器从1开始，每次循环增加3：

```
for (i = 1; i < 18; i += 3)
{ printf("%d ", i); }  /* Prints 1, 4, 7, 10, 13, 16 */
```

下面的代码将会产生有趣的结果：

```
for (outer = 1; outer <= 3; outer++)
  { for (inner = 1; inner <= 5; inner++)
    { printf("%d ", inner);
  }
  printf("\n"); /* a newline every 5 times */
}
```

下面是代码的输出：

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

如果把一个for循环放在另一个for循环的主体中，你就嵌套（nesting）了循环。实际上，内层的循环执行了外层循环指定的次数。如果想打印3张前5名顾客的列表，就可能需要嵌套的for循环。外层的循环从1到3，而内层的循环用于打印前5名顾客。

附录B中的21点游戏程序用了一个for循环来初始化52张牌。下面是代码：

```
for (sub = 0; sub <= 51; sub++)  {  /* Counts from 0 to 51 */
  val = (val == 14) ? 1 : val; /* If val is 14, reset to 1 */
  cards[sub] = val;
  val++;  }
```

你还没有学习非字符数组（一种特别的变量列表，如card[]），但是你可以看到for循环的计数器递增，sub的值从0开始直到等于51（一共52张牌）。带有?:运算符的那行代码确保val的值绝不会比14大。从1到13的4组数字（总数是52）被赋给初始的数组。从1到13的数字代表从Ace到King的牌。



奖励

- 想通过循环来递增或递减变量时，就可以使用for循环。
- 请记住for循环的关系测试在循环的顶部执行。
- 如果想循环特定的次数，就使用嵌套的循环。



陷阱

- 不要忘记for循环中的分号。for循环需要它们。
- 如果想让for循环的计数器递减，不要使用比测试值小的初始值。

15.3　小结

本章展示了C语言中另一种循环方式。for语句比while或do-while循环给你更多对循环的控制。for语句用变量来控制循环，变量被初始化并根据表达式被修改。

1. 代码范例

```
for (chap = 1; chap <= 15; chap++)
  { printf ("You're done with Chapter %d \n", chap); }
```

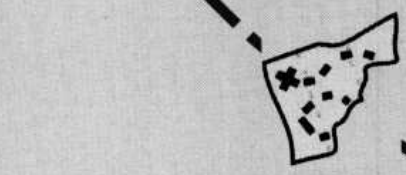
2. 代码分析

下面是代码产生的输出：

```
You're done with Chapter 1
You're done with Chapter 2
You're done with Chapter 3
:
```

依此类推，直到打印出you are done with Chapter 15为止。for语句使循环控制变量chap从初始值1递增到最终的15。一旦chap变得比15更大，for语句就结束循环，而且程序会继续向后执行。

16



C语言解释%
135



变量除以2
137



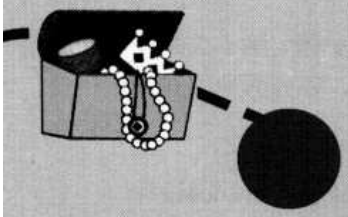
break很简单
134



只能在循环中使用continue
137



不要在没有某种关系测试的
情况下使用break或continue
137



第16章

终止循环

使用break和continue语句

本章内容

使用break语句
134

使用continue语句
135

本章不打算教你如何使用另一种循环。本章是对前面两章的扩展。除了关系测试，还可以用其他方式来控制while循环。除了使用计数器变量，还有其他方式来改变for循环执行的方式。

当你想提前退出循环或者想更快地进行下一次循环时，break和continue语句就可以帮你控制循环。

16.1　使用 break 语句

break语句很少单独出现在一行上。一般来说，break出现在if语句的主体中。原因很快就会告诉你。下面是break的语法格式：

```
break;
```



警告　break很简单，不是吗？是的。不过，还要记住break通常出现在if的主体中。换句话说，if几乎是每个break的第一部分。

break总是出现在循环中。break的作用是终止当前的循环。循环结束时，循环主体后面的代码就接着执行。break出现在循环的主体中时，break会立刻终止循环而程序的其他部分会继续执行。

下面的for循环正常情况下会打印出10个数字。不过，其中的break使得循环在打印了5个数字后就停止了。

```
for (i = 0; i < 10; i++)
  { printf("%d ", i);
    if (i == 4)
      { break; }  /* Quits loop after i becomes 4 */
}
/* Rest of program goes here */
```

来看一个现实的例子，假设一个教师写了一个程序来求25个学生测验成绩的平均分。下面的程序记录了25个学生的累积总分。但是，如果一两个学生没有参加测验，那么就不必按25个学生来求平均分了。如果教师输入-1.0作为测验成绩，就会触发break语句，而且循环会提前结束。

```
/* Test Scoring */
#include <stdio.h>
main()
{
  int numTest;
```

```
  float stTest, avg, total = 0.0;
  /* Asks for up to 25 tests */
  for (numTest = 0; numTest < 25; numTest++)
  { printf("What is the next student's test score? ");
    scanf(" %f", &stTest);
    if (stTest < 0.0)
      { break; }  /* Quits early if no more students */
    total += stTest;
  }
  avg = total / numTest;
  printf("\nThe average is %.1f%%.\n", avg);
  return 0;  /* Exits the program */
}
```

在讨论程序之前，先来看一下运行示例：

```
What is the next student's test score? 89.9
What is the next student's test score? 92.5
What is the next student's test score? 51.0
What is the next student's test score? 86.4
What is the next student's test score? 78.6
What is the next student's test score? -1

The average is 79.7%.
```

这一天有很多学生都病了！如果所有的25个学生都露面了，for循环将会确保要求输入25个测验成绩。然而，因为只有5个人参加了测验，教师不得不通过输入负数来告诉程序她已经完成了成绩的输入并且现在就想要得到5个学生测验成绩的平均分。



线索 为了在最后的平均分后面加上%号，在printf()的控制字符串中要用两个%。C语言把一个%看成是控制码，除非像这个程序那样把两个%%放在一起使用。不过，C语言仍然会把第一个%解释成第二个%的控制码。换句话说，%是它自己的控制码。



警告 break只是提前终止了while、do-while或for循环。break不能从if语句中退出，if不是循环语句。图16-1解释了break的行为。

16.2 使用 continue 语句

break使循环提前跳出（break），与之相反，continue强迫循环提前继续

(continue)。这也是它们被这样命名的原因！根据for、while或do-while循环的复杂性，你可能不想每次迭代都执行整个循环的主体。continue实际上表示："C语言，请忽略这次迭代剩余的主体部分。回到循环的顶部并开始下一次循环。"

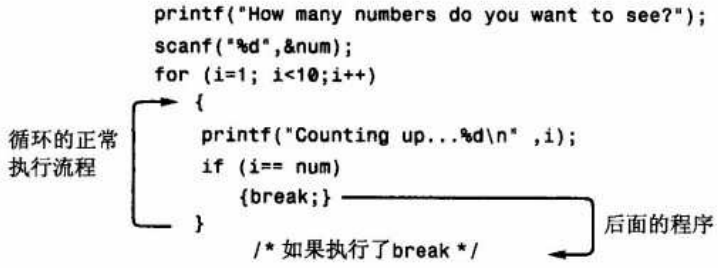


图16-1　break提前终止了循环



线索　词语迭代（iteration）是循环周期的计算机术语。程序员有时候觉得如果用了别人都不理解的词语，他们就能保住工作吧（笑话而已）。

下面的程序很好地展示了continue。程序包含一个for循环，它从1计数到10。如果循环变量包含一个奇数，就会打印消息I'm rather odd...，并且continue指示C语言忽略循环主体的剩余部分，因为它还需要为剩下的偶数打印Even up!。

```
/* Odd and even */
#include <stdio.h>
main()
{
  int i;
  for (i = 1; i <= 10; i++)
    { if ((i%2) == 1)  /* i is odd if true */
      { printf("I'm rather odd...\n");
        continue;
      }
      printf("Even up!\n");
    }
  return 0;  /* Exits the program */
}
```

下面是程序的输出：

```
I'm rather odd...
Even up!
I'm rather odd...
Even up!
I'm rather odd...
Even up!
I'm rather odd...
```

```
Even up!
I'm rather odd...
Even up!
```



说明 与break一样，continue的前面通常有某种类型的if语句。如果总是想使用continue，那么就根本不需要输入循环主体的余下部分了。要记住，只在某些循环周期中才使用continue。



请跳过这里，这有点儿难 再来看一下前面的程序是如何知道循环变量是奇数的。如果变量被2整除的余数为1，则是奇数；如果为0，则是偶数。



奖励

- 用break来提前终止for、while或do-while循环。
- 用continue来强迫循环进入新的循环周期。



陷阱

- 不要在前面没有某种关系测试的情况下使用break或continue。

16.3 小结

本章讲解了如何用break和continue语句更好地控制循环。while、do-while和for循环都能用break提前终止或者用continue提前继续。心想如果大多数循环都能自然结束就好了，不过，在有的情况下，仍然需要改变循环方式或提前停止循环。

1. 代码范例

```
/* Divisibility check */
#include <stdio.h>
main()
{
  int i;
  for (i = 1; i < 100; i++)
  {
    printf ("Testing %d\n", i);
    /* Checks if the number is divisible by 3 AND 4 */
```

```
    if ((i % 3 == 0) && (i % 4) == 0)
      { printf ("Found it!\n");
        break; }
    /* Checks whether the number is divisible by 3 */
    if (i % 3 == 0)
      { printf ("I am divisible by 3.\n");
        printf ("But that's only half the test!\n");
        continue; }
    /* Checks whether the number is divisible by 4 */
    if (i % 4 == 0)
      { printf ("I am divisible by 4.\n");
        printf ("One out of two isn't bad!\n");
        continue; }
    printf ("I'm not divisible by 3 or 4!\n");
  }
  return 0;
}
```

2. 代码分析

上面的代码从1循环到100，并且报告第一个能同时被3和4整除的数。如果数字只能被3或4整除，就会打印消息，并进入下一个循环周期（用continue）。如果找到了能同时被3和4整除的数，循环顶部的break语句就会强迫for语句提前结束。

17



重新安排case语句
145



switch语句的使用
143



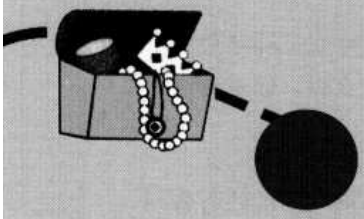
使用exit()函数
143



default像else那样工作
144



不要使用嵌套的if语句
145



第 17 章

测试多个值

switch语句

本章内容

使用switch语句
142

switch语句中使用break语句
144

妙用case语句
144

趣闻轶事
当创建要求用户输入的菜单时，也就是创建用户界面。

if语句对简单的数据测试很好用，尤其是在数据测试只有两种或三种结果的情况下。你也可以用if语句来测试多于两个值的情况，但如果要这么做，你就不得不嵌套多个if语句了，而且这样写容易混乱并且难以维护。

想一下你是如何根据用户对菜单的响应来执行代码的。菜单是供人选择的选项列表，比如下面这个菜单：

```
What do you want to do?
  1. Add information
  2. Change information
  3. Print information
  4. Delete information
  5. Quit the program
What is your choice?
```

需要用4条嵌套的if-else语句才能处理所有的条件，如下所示：

```
if (userAns == 1)
  { /* Perform the add routine */ }
else
  if (userAns == 2)
    { /* Perform the change routine */ }
  else
    if (userAns == 3)
      { /* Perform the print routine */ }
    else
      if (userAns == 4)
        { /* Perform the delete routine */ }
      else
        { /* Perform the quit routine */ }
```

使用嵌套的if语句并没有错，但是在多种条件的情况下使用switch语句会更清晰。

17.1　使用 switch 语句

switch的语法格式是C语言（或者任何其他语言）中最长的一种格式。下面是switch的语法格式：

```
switch (expression)
  { case (expression1): { one or more C statements; }
    case (expression2): { one or more C statements; }
    case (expression3): { one or more C statements; }
      /* If there are more case statements,
         they would go here  */
    default: { one or more C statements; }
}
```



请跳过这里，这有点儿难 与大多数语句一样，switch语句的实际用法并不像它的语法格式那么令人生畏。

前面展示的菜单非常适合通过调用一系列的函数来实现。问题是本书还没有讨论函数调用，只涉及了少量的内置函数，如printf()和scanf()。下面简单的switch语句根据用户的选择打印出相应的消息。通常，函数调用会取代每个case后面的printf()语句。在读完第31章后，你将会理解如何使用函数调用来执行case的行为。

```
/* Menu printing statements go here */
do
{  printf("What is your choice? ");
  scanf(" %d", &choice);
  switch (choice)
  { case (1) : printf("You are adding.\n");
               break;
    case (2) : printf("You are changing.\n");
               break;
    case (3) : printf("You are printing.\n");
               break;
    case (4) : printf("You are deleting.\n");
               break;
    case (5) : exit(1);  /* Quits program.
                         Requires stdlib.h */
               break;
    default  : printf("I don't know ");
               printf("option %d.\n", choice);
               printf("Try again.\n");
               break;
  }
} while ((choice < 1) || (choice > 5));
/* Rest of program would follow */
```

case语句根据choice的值来决定行为的路线。例如，如果choice等于2，就会打印出消息You are changing.。如果choice等于5，程序就会用内置的exit()函数退出。



警告 无论何时，当你需要在程序自然结束之前终止它时，就使用exit()函数。exit()括号中的值将返回给操作系统。大多数初学者忽视返回值，在括号里随便写上0或1。使用了exit()函数的程序必须有include <stdlib.h>。

do-while循环保证了用户必须输入1到5之间的数字。如果不是，由于有

default关键字，就会打印出消息I don't know。C语言保证了如果没有一种情况与switch后面的变量相匹配，default的语句就会执行。



说明　default的工作方式与else有点相象。如果if测试为假，else后面的语句就会执行，而如果没一个case条件与switch变量相匹配，default后面的语句就会执行。尽管default是可选的（和else一样），但是用default来处理没有预料到的switch值是一种良好的编程习惯。



线索　switch变量可以是整型或字符型变量。不要用float型或double型变量进行switch测试。

17.2　switch 语句中使用 break 语句

前面展示的switch语句中有几条break语句分散于代码中。break确保了只有一个case会执行。没有break语句，switch就会穿过其他的case语句。如果从switch中去掉break语句且用户输入choice的值为2，程序的输出如下所示：

```
You are changing.
You are printing.
You are deleting.
```

break避免了switch的case语句一起执行。



说明　default条件的消息没有被打印出来的唯一原因是在case (5)中执行了exit()函数。

17.3　妙用 case 语句

case语句不必按照任何顺序排列。即使是default也不必排在最后一个。事实上，只要default出现在switch的最后面，就不需要在default后面加上break。然而，如果要重新安排case语句，在default后面加上break能确保移动了全部语句。如果把default放在case语句的前面，就需要加上break以避免执行其他的case语句。



线索 你可以重新安排case语句以提高效率。把最常用的case放在switch语句的顶部，这样C语言就不需要往下搜索以寻找匹配的case语句了。

附录B中21点游戏程序的dispCard()函数使用switch语句来打印纸牌并且更新每方的分数。



奖励

- 用switch语句对菜单选择和其他类型的应用（需要从多个值中做选择）进行编程。
- switch语句中只能使用整型或字符型变量，因为float和double型的值不能准确匹配。
- 如果不想让下面的case语句执行，就在每个case后面加上break。



陷阱

- 可以使用switch语句时，不要使用嵌套的if语句。switch是更清晰的语句。

17.4 小结

本章讲解了C语言的switch语句。switch分析一个整型或字符型变量的值并执行几个代码段（称为case）中的一个。你可以用嵌套的if语句编写等价的代码，但是switch语句更清晰——尤其是当程序需要分析用户对菜单的响应并执行对应的代码段时。

1. 代码范例

```
/* Tax code display */
#include <stdio.h>
main()
{
  char choice;
  printf("Are you filing a single, joint, or ");
  printf("married return (s, j, m)? ");
  do
    { scanf(" %c", &choice);
      switch (choice)
      { case ('s') : printf("You get a $1,000 deduction.\n");
                     break;
        case ('j') : printf("You get a $3,000 deduction.\n");
```

```
                    break;
        case ('m') : printf("You get a $5,000 deduction.\n");
                    break;
        default    : printf("I don't know the ");
                    printf("option %c.\n", choice);
                    printf("Try again.\n");
                    break;
    }
  } while ((choice != 's') && (choice != 'j') &&
                              (choice != 'm'));
  return 0;
}
```

2. 代码分析

这段代码首先询问用户一个具有3种可能答案的问题。根据回答的结果s、j或m，程序打印出相应的与税率相关的消息。

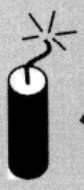
编写switch语句时应该总是处理未预期的异常。如果用户输入除了s、j或m之外的任何东西，do-while循环会确保始终向用户提问，直到用户输入了它们其中的一个。

如果你自己测试程序，请确保输入一个小写的答案，于是switch就可以用合适的case来匹配你的答案了。

18



第2个for循环继续
152



getch()不会在屏幕
上回显输入的字符
154



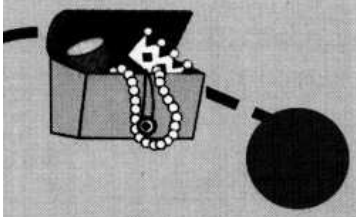
使用int就会安全
152



学了break，难道你不高兴吗
152



不要和字符变量一起
使用字符I/O函数
154



第 18 章

输入和输出

使用内置I/O函数

本章内容

putchar()和getchar()函数
150

关于换行符的思考
152

getch()函数
153

除了使用scanf()和printf()函数之外，还有其他输入和输出的方式。本章介绍一些C语言的内置I/O函数，这些函数可以用来控制I/O。你可以用这些简单的函数构建出强大的数据输入程序。

这些函数提供了一次输入或输出一个字符的原始的（primitive）功能。当然，你也可以用scanf()和printf()函数加上%c格式来处理一个字符。然而，这里要介绍的字符I/O函数更容易使用，并且它们提供了一些scanf()和printf()函数没有的功能。

18.1　putchar()和getchar()函数

getchar()函数从键盘上读取一个字符，而putchar()函数把一个字符发送到屏幕上。图18-1说明了使用这些函数时所发生的一切。它们基本上按照你所认为的方式工作。无论何时当你想打印或者读取一个字符到变量中时，都可以使用它们。

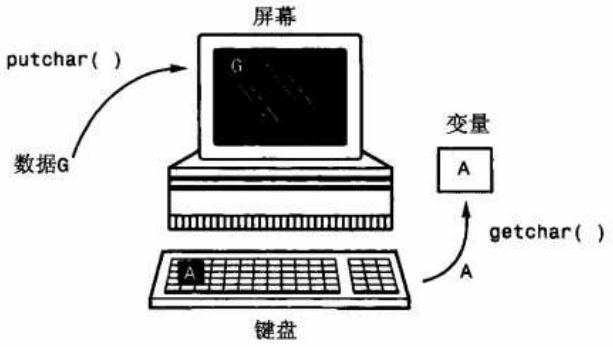


图18-1　getchar()和putchar()用于输入和输出一个字符



线索　当使用本章介绍的I/O函数时，始终要包含STDIO.H头文件，和使用printf()和scanf()时一样。



说明　正如它们名字的含义一样，getchar()是“获取字符”，而putchar()是“发送字符”。C语言的设计者知道他们在干什么！

下面的程序打印C is fun，一次打印一个字符，用putchar()依次打印字符数组的元素。请注意strlen()被用来确保for循环的输出不会超过字符串的末尾。

```
#include <stdio.h>
#include <string.h>
main()
{
  int i;
  char msg[] = "C is fun";
  for (i = 0; i < strlen(msg); i++)
    { putchar(msg[i]); }  /* Outputs a single character */
        putchar('\n');    /* Must do because putchar() doesn't */
              return 0;   /* All done */
}
```

getchar()函数返回从键盘输入的字符。因此，你通常把字符赋给一个变量或用它做别的事。你可以把getchar()单独写成一行，像下面这样：

```
getchar();  /* Does nothing with the character you get */
```

但是大多数C语言编译器会警告你这条语句无用。getchar()函数会从键盘上获取一个字符，但是随后并不用这个字符来做任何事情。



警告 附录B中21点游戏程序的getAns()函数有一行只写了getchar()。不管你是否相信，这条语句并不会与上一段落的内容产生矛盾。一会儿你就会明白。

下面的程序一次从键盘上读取一个字符并且把字符存储在一个字符数组中。随后，用一连串的putchar()从后面开始打印数组。

```
#include <stdio.h>
#include <string.h>
main()
{
  int i;
  int msg[25];
  printf("Type up to 25 characters then press Enter...\n");
  for (i = 0; i < 25; i++)
    { msg[i] = getchar();  /* Gets a character at a time */
      if (msg[i] == '\n')
        {
          i--;
          break; }  /* Quits if user presses Enter */
    }
  putchar('\n');  /* Prints a blank line */
  for (; i >= 0; i--)
    { putchar(msg[i]); }  /* Prints a character at a time */
  putchar('\n');
  return 0;  /* All done */
}
```



线索　请注意第2个for循环的变量i没有写初始值！实际上，它有。i的值在上一个for循环之后变为了最后一个数组下标。因此，第2个for循环继续从第1个for循环留下的i值开始。



警告　getchar()函数的输入字符一般定义为int型，就像这个例子一样。在C语言的数据类型中，只有整型和字符型数据不需要强制类型转换就能交互使用。在一些高级应用中，getchar()可能会返回一个char数据类型无法使用的值，因此用int就安全了。



说明　难道你不为学习了break而兴奋吗？程序将一直一次读取一个字符，直到用户按下Enter键（它会产生一个换行符\n转义序列），然后break语句就会终止循环。

18.2　关于换行符的思考

尽管getchar()读取一个字符，但是控制不会回到程序中直到用户按下Enter键。实际上，getchar()函数指示C语言把输入接收到缓冲区（buffer）中，缓冲区是一块为输入预留的内存区域。缓冲区不会释放，直到用户按下Enter键，然后缓冲区的内容才以一次一个字符的方式释放。这意味着两件事情。第一，只要用户还没有按下Enter键，他就可以用Backspace键来纠正错误的字符输入。第二，如果你没有去除Enter键的按下，它就会留在输入缓冲区中。

去除Enter键的按下对于所有C语言初学者来说都是一个问题。有几种办法，但是都不够优雅。看一看下面的程序段：

```
printf("What are your two initials?\n");
firstInit = getchar();
lastInit  = getchar();
```

你可能会认为如果用户输入GT，G会保存在变量firstInit中而T会保存在变量lastInit中，但事实并非如此。直到用户按下Enter键，第一个getchar()才完成，因为G被送到缓冲区中。只有当用户按下Enter键时，G才会离开缓冲区并进入程序中。但这时，Enter仍然在缓冲区中！因此，第二个getchar()把Enter（实际上，是代表Enter的\n）发送到lastInit中！T仍然留在缓冲区中以供下一个getchar()（如

果有的话）获取。



线索 解决这个问题的一个方法是插入另外一个getchar()来捕捉Enter，但是不用它做任何事。这就是为什么附录B中21点游戏程序有一行单独的getchar()——用来捕捉Enter并清空缓冲区为下一次字符输入留出空间。

下面是获取初值问题的程序：

```
printf("What are your two initials?\n");
firstInit = getchar();
nl = getchar();   /* Grabs the newline */
lastInit  = getchar();
nl = getchar();   /* Grabs the newline */
```

这段代码要求用户在每两个初值之间按下Enter键。你不需要对变量nl做任何事情，因为nl只是用来存放换行符的。像21点游戏所做的那样，你甚至不需要把换行符放在变量中。下面的代码与上面的一样工作：

```
printf("What are your two initials?\n");
firstInit = getchar();
getchar();   /* Discards the newline */
lastInit  = getchar();
getchar();   /* Discards the newline */
```

当编译包含单独的getchar()语句的程序时，一些C编译器会发出警告消息。只要你是用这些getchar()来捕捉Enter键的按下，你就可以忽略编译器的警告。

你也可以通过要求在用户输入两个初值之后按下Enter键来获取两个初值，像这样：

```
printf("What are your two initials?\n");
firstInit = getchar();
lastInit  = getchar();
getchar();   /* Discards the newline */
```

如果用户输入GP，然后按下Enter键，那么G会存放在firstInit变量中而P会存放在lastInit变量中。

18.3 getch()函数

名为getch()的字符输入函数有助于消除getchar()余留下的Enter键的按下。

getch()是*无缓冲的*（unbuffered）——getch()即时获取用户输入的任何按键而且不等待Enter键的按下。getch()的缺点是用户不能用Backspace键来纠正错误的输入。例如，用getchar()时，如果用户想输入D却错误地输入了B，他可以按下Backspace键来改正。B将被从缓冲区中取走，而一旦按下Enter，getchar()就将获取到D。因为getch()没有对输入进行缓存，也就没有机会用Backspace了。下面的代码用于获取两个字符，而不需要按下Enter键：

```
printf("What are your two initials?\n");
firstInit = getch();
lastInit  = getch();
```

getch()比getchar()更快一点，因为它不需要等待用户按下Enter键。因此，不需要像getchar()那样用单独的getch()来去除\n。



请跳过这里，这有点儿难　getch()不会像getchar()那样把输入的字符回显在屏幕上。因此，如果你想让用户看到他输入的字符，就需要在getch()后面加上putch()。要回显初值，可以这样做：

```
Printf("What are your two initials?\n");
firstInit = getch();
putch(firstInit);
lastInit = getch();
putch(1astInit);
```

下一章将讲解更多的内置函数，包括两个快速输入和输出字符串的函数，它们使用起来与本章介绍的I/O函数处理字符一样容易。



奖励

- 用getchar()和putchar()来输入和输出一个字符。
- 用单独的getchar()来去除Enter键的按下。你也可以像范例代码一样，创建一个循环来调用getchar()，直到返回值是\n。
- 用getch()来获取无缓冲的一个字符，只要用户一输入就能获取到。



陷阱

- 不要和字符变量一起使用字符I/O函数。要用int型变量。
- 如果想让用户输入的内容回显在屏幕上，不要忘了用putch()来打印字符。

18.4 小结

本章讲解了其他一些输入和输出函数。这里给出的是字符I/O函数。与scanf()和printf()不同，getchar()、getch()、putchar()和putch()函数一次输入和输出一个字符。

你会经常发现getch()和getchar()函数用于大型输入程序中。这意味着这些函数一次从键盘上获取一个字符并把字符加到数组中。

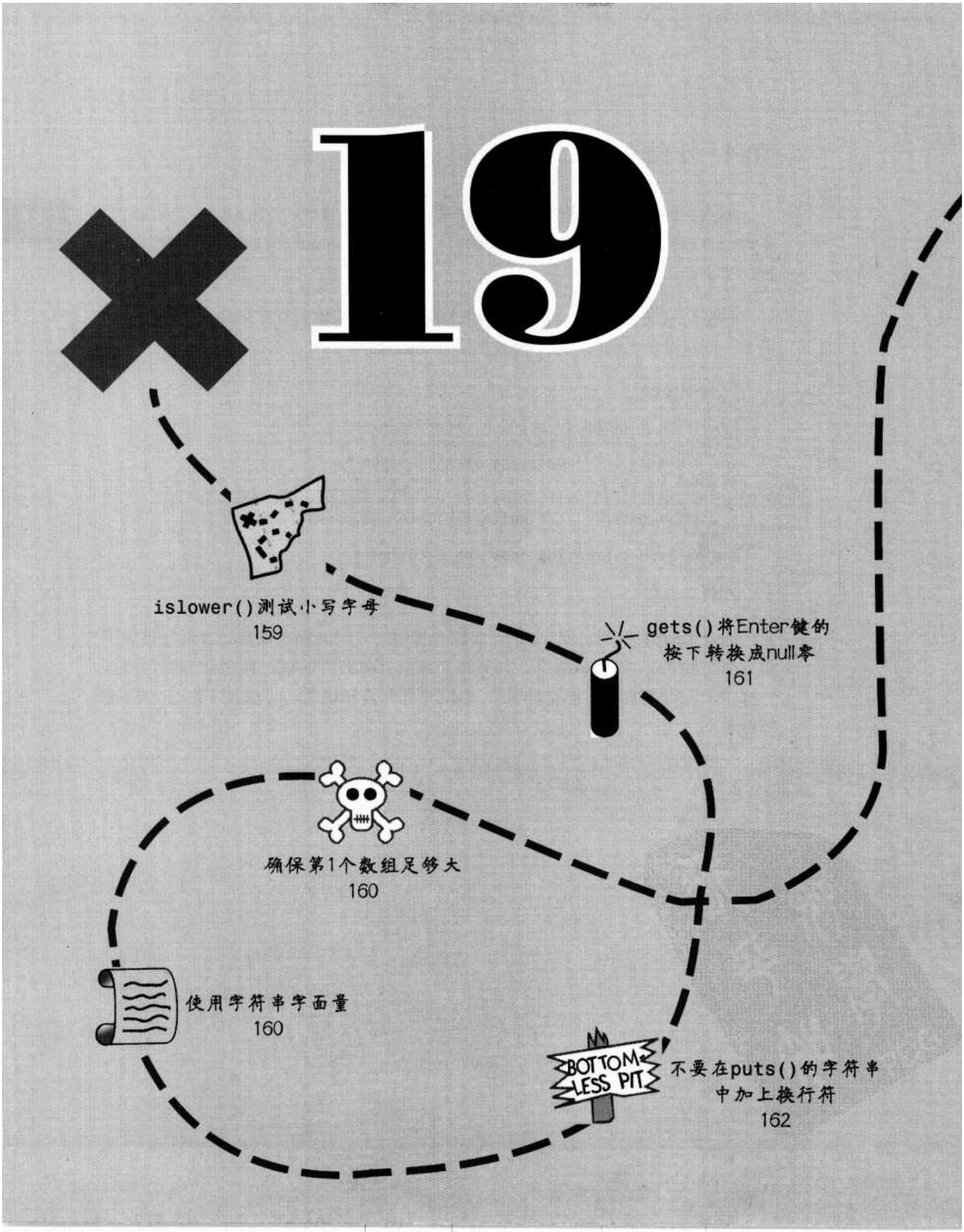
1. 代码范例

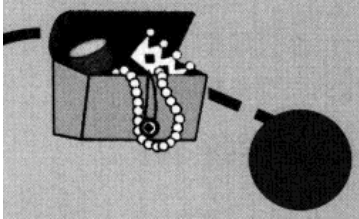
```
printf("I'm going to print a report now.\n");
printf("Is your printer turned on (y/n)? ");
ans = getch();  /* Gets the user's answer */
if (ans == 'n')
  { printf("I'll wait. Press Enter when you have it on...\n");
    ans = getch();  /* Waits for Enter keypress */
}
/* The report printing code would follow */
```

2. 代码分析

这段代码向用户提了一个是非问题并等待用户用y或n来响应。if语句确保了如果用户输入n，就打印消息。如果用户还没有打开打印机，程序会等待用户打开打印（否则报表就不能被打印）。当用户打开打印机后，可以按下Enter键来通知程序。

19


islower()测试小写字母
159
gets()将Enter键的
按下转换成null零
161
确保第1个数组足够大
160
使用字符串字面量
160
BOTTOM LESS PIT
不要在puts()的字符串
中加上换行符
162



第 19 章

更多有关字符串的内容

C语言内置的字符函数和字符串函数

本章内容

字符测试函数
158

大小写测试函数
158

大小写转换函数
159

字符串函数
159

本章再教你一些非常有用的C语言内置函数。除了目前为止你已经学到的strlen()、getchar()和printf()函数，C语言还包含了很多有用的内置函数。

趣闻轶事
你的电脑可能还有其他一些有用的函数。查看它的文档。

用一章的篇幅也不能列出全部的内置函数。本章讲解最常用的和最有用的字符和字符串函数。下一章，你将学到一些数值函数。

19.1　字符测试函数

有几种内置的字符测试函数（character-testing function）。你已经知道如何使用getchar()和getch()来获取一个字符了，而字符测试函数可以用来准确判断程序接收到输入字符的类型。你可以根据字符测试的结果，设置if逻辑来执行特定的行为。



线索　使用这里介绍的字符函数时，都要在程序的顶部包含CTYPE.H头文件。

isalpha()函数返回真（对于C语言来说是1），如果括号里的值是从a到z（或大写的A到Z）的字母。而如果括号里的值是任何其他字符，就返回假（对于C语言来说是0）。看一看下面的if语句：

```
if (isalpha(inChar))
  { printf("Your input was a letter.\n"); }
```

只有当inChar的值是一个字母时，才会打印出消息。

C语言还有一个相应的名为isdigit()的函数，如果括号里的字符是从0到9的数字，它就返回真。如果inChar的值是一个数字，下面的if语句会打印A number。

```
if (isdigit(inChar))
  { printf("A number\n"); }
```



说明　你知道它们为什么叫做字符测试函数吗？isalpha()和isdigit()都测试字符的内容并返回测试的结果。

19.2　大小写测试函数

isupper()和islower()函数使你知道一个变量是否包含了大写或小写的值。使用isupper()使得你不用写像下面这样很长的if语句：

```
if ((inLetter >= 'A') && (inLetter <= 'Z'))
  { printf("Letter is uppercase\n"); }
```

只需使用isupper()而不需要编写上面复杂的逻辑比较了：

```
if (isupper(inLetter))
  { printf("Letter is uppercase\n"); }
```




线索 islower()与isupper()测试大写字母一样，用来测试小写字母。

你也许想用isupper()来确保用户在输入名字时输入大写字母。

19.3 大小写转换函数

有两个重要的字符转换函数（也叫做字符映射函数），它们把参数稍候修改再返回。与isupper()和islower()只是测试字符的值并返回真或假不同，toupper()和tolower()把它们的参数转换成不同的大小写状态并返回。toupper()返回参数的大写形式。tolower()返回参数的小写形式。

下面的程序段根据用户的输入打印yes或no。不用toupper()函数，代码会很长。

```
if ((userInput == 'Y') || (userInput == 'y'))
  { printf("yes\n"); }
else
  { printf("no\n"); }
```

下面的程序段用toupper()简化了if语句的逻辑测试：

```
if (toupper(userInput) == 'Y') /* No need to */
  { printf("yes\n"); }         /* test lower */
else
  { printf("no\n"); }
```



线索 21点游戏程序在getAns()函数中用toupper()返回用户输入的大写形式：

```
return toupper(ans);
```

19.4 字符串函数

STRING.H头文件不仅仅包含了strcpy()和strlen()的函数描述。本节讲解

strcat()函数，它能够合并两个字符数组，只要数组存放了字符串。strcat()函数代表字符串连接（string concatenation）。

strcat()函数把一个字符串附加到或加到另一个字符串的末尾。下面的代码段说明了strcat()函数的功能：

```
char first[25] = "Peter";
char last[25] = " Parker";
strcat(first, last);  /* Adds last to the end of first */
printf("I am %s\n", first);
```

下面是这段代码的输出：

```
Peter Parker
```

strcat()函数需要两个字符串参数。strcat()把第2个字符串附加到第1个字符串的末尾。在姓的前面出现空格的原因是因为在第2行中，last数组初始化时，在姓的前面有一个空格。



警告　你得确保第1个数组足够大以存放两个字符串。如果你试图把一个字符串连接到另一个字符串的后面，而第2个字符串没有定义足够的空间大小来存放两个字符串，那么就会产生奇怪且不可预知的结果。



说明　因为strcat()函数的第2个参数没有修改，所以如果你愿意的话可以使用字符串字面量，而不是字符数组。

puts()和gets()函数提供了一种更容易的方法来打印和获取字符串。它们的描述在STDIO.H头文件中，所以你不需要为puts()和gets()函数增加额外的头文件。puts()把字符串发送到屏幕，而gets()从键盘获取字符串。下面的程序论证了gets()和puts()。请注意这里我们不是用printf()和scanf()来输入和打印字符串。

```
/* Get and put strings */
#include <stdio.h>
#include <string.h>
main()
{
  char city[15];
  char st[3];  /* Leave 1 for null zero */
  char fullLocation[18] = "";  /* Put empty string here */
  puts("What town do you live in? ");
```

```
    gets(city);
    puts("What state do you live in? (2-letter abbreviation)");
    gets(st);
    /* Concatenates the strings */
    strcat(fullLocation, city);  /* Adds a comma and a space */
    strcat(fullLocation, ", ");  /* to end of city */
    strcat(fullLocation, st);  /* Adds the state
                                  to end of city */
    puts("\nYou live in ");
    puts(fullLocation);
    return 0;
}
```




线索 不得不3次使用strcat()：一次增加城市名，一次增加逗号，一次增加州名。

下面是程序的一个输出：

```
What town do you live in?
Jacksonville
What state do you live in? (2-letter abbreviation)
FL

You live in
Jacksonville, FL
```



线索 puts()会自动在打印的每个字符串末尾加上换行符。你不需要在输出字符串的后面加上\n，除非你想打印出一个空行。



请跳过这里，这有点儿难 gets()会把Enter键的按下转换成null零，以确保从键盘获取的数据是以null结尾的字符串，而不是一个字符组成的数组。

用gets()而不用scanf()的一个最主要的原因是用户输入的字符串可以包含嵌入的空格，比如全名（名和姓）。而scanf()不能接受带有空格的字符串，它在遇到第一个空格时就会停止接收用户的输入。



奖励

- 使用C语言内置的字符测试和字符映射函数，这样程序就不用费力地判断字符数据的大小写了。

- 用gets()函数来获取字符串，用puts()函数来打印字符串。
- 当你必须获取带有空格的字符串时，要用gets()函数。请记住scanf()函数不能获取带有空格的字符串。
- 用strcat()函数来合并两个字符串。



陷阱

- 如果你不能肯定第1个字符数组可以存放合并后的字符串，就不要用strcat()来连接两个字符串。
- 不要在puts()的字符串中加上换行符，除非你想打印出一个空行。puts()会自动在字符串的末尾加上换行符。

19.5　小结

本章讲解了一些内置的字符和字符串函数，这些函数可以测试和修改字符串。字符串是字符的字面量列表，或者显式地声明在引号中或者存储在字符数组中。本章介绍的字符串函数工作在字符串字面量和数组上。

有这样一些函数，包括测试字符是否是数字和字母，把大写的和小写的字符转换成相反的形式，连接（合并）字符串，以及允许快速输入和输出字符串。

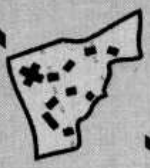
1. 代码范例

```
/* Convert to upper and lowercase */
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
main()
{
  char name[25], nameNew[25];
  char ans;
  int i;
  printf("What is your name? ");
  gets(name);
  printf("Are you sure that %s is your name (y/n)? ", name);
  ans = getch();
  if (toupper(ans) == 'Y')
    { for (i = 0; i <= strlen(name); i++)
        { nameNew[i] = toupper(name[i]); }
      printf("Your name in uppercase letters is %s.\n", nameNew);
    }
  return 0;
}
```

2. 代码分析

这段代码首先要求用户输入他的名字。名字是用gets()函数输入。用户可以输入名和姓，因为用的是gets()函数，而不是scanf()函数。如果用户核实名字输入正确，for语句把名字中的每个字符都转换成大写字母。for循环逐个访问数组的元素，把每个字符发送给toupper()函数并把大写字母存储在名为nameNew的数组中。

20



不能计算负数的平方根
167



熟悉数学符号
167



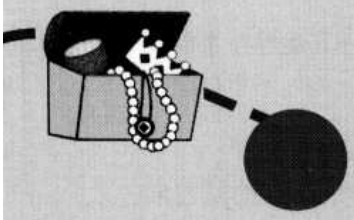
每个函数都返回一个
float类型的值
166



负值有意义
167



你现在还不必掌握三角
函数和对数函数
171



第20章

更高级的数学函数

C语言内置的数值函数

本章内容

简单的数学函数
166

更多的转化
167

三角函数和对数函数
168

获取随机数
169

本章将介绍内置函数中的数值函数。C语言能帮助你解决一些其运算符不能单独完成的数学问题。最重要的是，C语言的内置数值函数提供了一些你不用自己再去编写的函数。

C语言的许多内置数学函数非常专业化。它们的使用并不难，难的是它们的用途。除非需要三角法的和高级的数学函数，你可能不会用到本章介绍的许多函数。



线索　有些人写了很多年的C程序，而从来不需要用到这些函数。你应该阅读本章的内容以了解C语言能做些什么，这样当你确实要用到这些强大的函数时，你就知道哪些对你有用了。

20.1　简单的数学函数

本章描述的所有函数要求使用MATH.H头文件。如果使用数学函数，要确保包含了MATH.H和STDIO.H头文件。首先介绍的一些数学函数没有太多的数学功能，因为它们是数值函数。这些函数把一些数字转化成其他数字。

`floor()`和`ceil()`函数分别称为上舍入和下舍入函数。它们把非整数变成与其相邻的更小的或更大的整数值。例如，如果想计算找回的零钱（包含美元和美分）中有多少美元的钞票，你可以使用`floor()`函数。如下面的代码所示：

```
change = amtPaid - cost;  /* Floating-point values */
dollars = floor(change);
printf("The change includes %f dollar bills.\n", dollars);
```



警告　尽管`ceil()`和`floor()`函数把参数转化成整数，但是每个函数都返回`float`值！这也是用`%f`转化代码打印变量`dollars`的原因。

`ceil()`函数（与`floor()`相反的函数）上舍入某浮点数最接近的整数。`ceil()`和`floor()`函数也都能处理负值，如下面的代码所示：

```
lowVal1 = floor(18.5);   /* Stores  18.0 */
lowVal2 = floor(-18.5);  /* Stores -19.0 */
hiVal1 = ceil(18.5);     /* Stores  19.0 */
hiVal2 = ceil(-18.5);    /* Stores -18.0 */
```



说明 当考虑负数的方向时，负值就有了意义。-18.5下面的整数是-19。-18.5的上一个整数是-18。
看，这些函数不算太差，它们在需要的时候提供了方便。

20.2 更多的转化

另外有两个数值函数把数字转化成其他值。fabs()函数返回浮点绝对值(absolute value)。第一次听说绝对值时，听起来似乎永远不需要用到它。无论是正数或负数，一个数字的绝对值，都是正数形式。下面的两个printf()函数都打印25：

```
printf("Absolute value of 25.0 is %.0f.\n", fabs(25.0));
printf("Absolute value of -25.0 is %.0f.\n", fabs(-25.0));
```



说明 打印出来的浮点数没有带小数点，因为在%f转化代码中出现了.0。



线索 在计算年龄、体重和距离的差别时，绝对值非常有用。例如，两个人的年龄差总是正数，无论用哪个减去哪个。

即使你不编写繁杂的科学或数学程序，还有两个数学函数也很有用。pow()函数计算一个值的幂，而sqrt()函数返回一个值的平方根。



线索 你不能计算负数的平方根。fabs()函数可以帮助你确保不会去计算负数的平方根，方法是在计算平方根之前把数字转化为正值。



请跳过这里，这有点儿难 也许一张图片就能带回关于熟悉的高中代数的记忆。图20-1表明了pow()和sqrt()所对应的熟悉的数学符号

下面的代码打印出10的3次幂和64的平方根：

```
printf("10 raised to the 3rd power is %.0f.\n",
       pow(10.0, 3.0));
printf("The square root of 64 is %.0f.\n", sqrt(64.0));
```

下面是这些printf()函数的输出：

```
10 raised to the 3rd power is 1000.
The square root of 64 is 8.
```

C程序员这样写：	数学家这样写：
`x = pow(4, 6);`	$x = 4^6$
`x = sqrt(value);`	$x = \sqrt{value}$

图20-1　查看pow()和sqrt()对应的数学符号

20.3　三角函数和对数函数

只有一些读者需要三角函数和对数函数。所以，如果你不会用也不想用它们，也可以跳过本节。如果你现在需要用到它们，也一定不想要太多的讲解，所以这里不作过多解释。

表20-1列出了主要的三角函数。每个函数都需要一个用弧度表示的参数。

表20-1　C语言的三角函数

函　数	含　义
cos(x)	返回角x的余弦
sin(x)	返回角x的正弦
tan(x)	返回角x的正切



请跳过这里，这有点儿难　如果想用度数而不是弧度来表示参数，你可以用下面的公式进行转换：

```
radians = degrees * (3.14159/180.0);
```

表20-2列出了主要的对数函数。

表20-2　C语言的对数函数

函　数	含　义
exp(x)	返回e^x的值
log(x)	返回x的自然对数，数学上写作ln(x)。x必须为正值
log10(x)	返回x以10为基的对数，数学上写作$\log_{10}(x)$。x必须为正值

下面的程序打印出6个三角函数和对数函数的结果：

```
/* Trig and log functions */
#include <stdio.h>
#include <math.h>
main()
{
  printf("The trig functions:\n");
  printf("  The cosine of 1 is %.3f\n", cos(1));
  printf("  The sine of 1 is %.3f\n", sin(1));
  printf("  The tangent of 1 is %.3f\n", tan(1));
  printf("The log functions:\n");
  printf("  e raised to 1 is %.3f\n", exp(1));
  printf("  The natural log of 2 is %.3f\n", log(2));
  printf("  The base-10 log of 2 is %.3f\n", log10(2));
  return 0;
}
```

下面是程序的输出。C语言计算这些值是否比你用纸和笔算得快？

```
The trig functions:
  The cosine of 1 is 0.540
  The sine of 1 is 0.841
  The tangent of 1 is 1.557
The log functions:
  e raised to 1 is 2.718
  The natural log of 2 is 0.693
  The base-10 log of 2 is 0.301
```

20.4 获取随机数

对于游戏和模拟程序，你经常需要产生随机数值。C语言的内置rand()函数正是完成这项工作的。它返回从0到32767的一个随机数。rand()函数要求添加STDLIB.H（标准库，standard library）头文件。如果想让随机数的取值范围变小，可以用%(求余运算符)来实现。下面的表达式把从1到6的随机数赋给变量dice：

```
dice = (rand() % 5) + 1;  /* From 1 to 6 */
```



警告 因为骰子的值可以从1到6，求余运算符返回整数除法的余数（从0到5），然后加上1以产生骰子的值。

如果你想要真正的随机数值，必须做一件疯狂的事。下面要会介绍。



说明　在程序每一次运行时，你可能总是想要产生一个不同的随机数集合。游戏需要这样的随机性。然而，许多模拟和科学研究需要重复相同的随机数集合。如果没有设置随机数产生器的种子，rand()将总是产生相同的随机数集合。

设置随机数产生器的种子意味着给它一个初始的基数值，rand()用这个值产生随机数。用srand()来设置随机数产生器的种子。每次运行程序时，srand()的括号里的数字必须不相同，除非你想产生相同的随机数集合。

有一个技巧可以在每次运行时都srand()一个不同的数字，就是在srand()的括号里写上准确的时间。电脑能精确到百分之一秒的时间。

因为没有时间进行详细的介绍，让我们直接了当地看看大多数程序员是如何产生真正的随机数的。下面的代码与附录B中21点游戏里的一段很类似。其中的注释解释了此时要做的事情。

```
time_t t;  /* Goes with your variable definitions
              such as int i;                        */
srand(time(&t));                  /* Seeds with the time of day */
subDraw = (rand() % (numCards));  /* Ensures that a random
                                     card is drawn from a
                                     52-card deck          */
```



线索　像这样用时间作为随机数产生器的种子之前，必须包含TIME.H头文件。



说明　本质内容是：如果你加了这两条看起来怪异的时间语句，rand()将总是产生随机数并且每次运行程序都会产生不同的结果。



奖励

- 尽量使用内置的数值函数，这样就不需要编写代码来实现相同的计算。
- 许多数值函数，如floor()、ceil()和fabs()，把一个数字转化成另一个数字。
- 如果想让rand()在每次运行程序时都产生不同的随机数，就用时间作为随机数产生器的种子。



陷阱

- 如果你现在不需要三角函数和对数函数，可以不必掌握它们。许多C程序从来不用它们。
- 不要使用整型变量来存放本章介绍的数学的返回值(除非对函数的返回值进行强制类型转换)，因为尽管一些函数（如`ceil()`）产生整数的结果，但是它们返回`float`或`double`型的值。

20.5 小结

本章讲解了许多内置的数学函数，它们使数值数据的处理变得更容易。C语言包含了各种整数函数、数值转换函数、时间和日期函数以及随机数产生函数。

你不需要现在就理解本章介绍的每一个函数。你可能编写成百上千的C程序，也不会用到这些函数。然而，如果你需要它们，C语言中也提供。

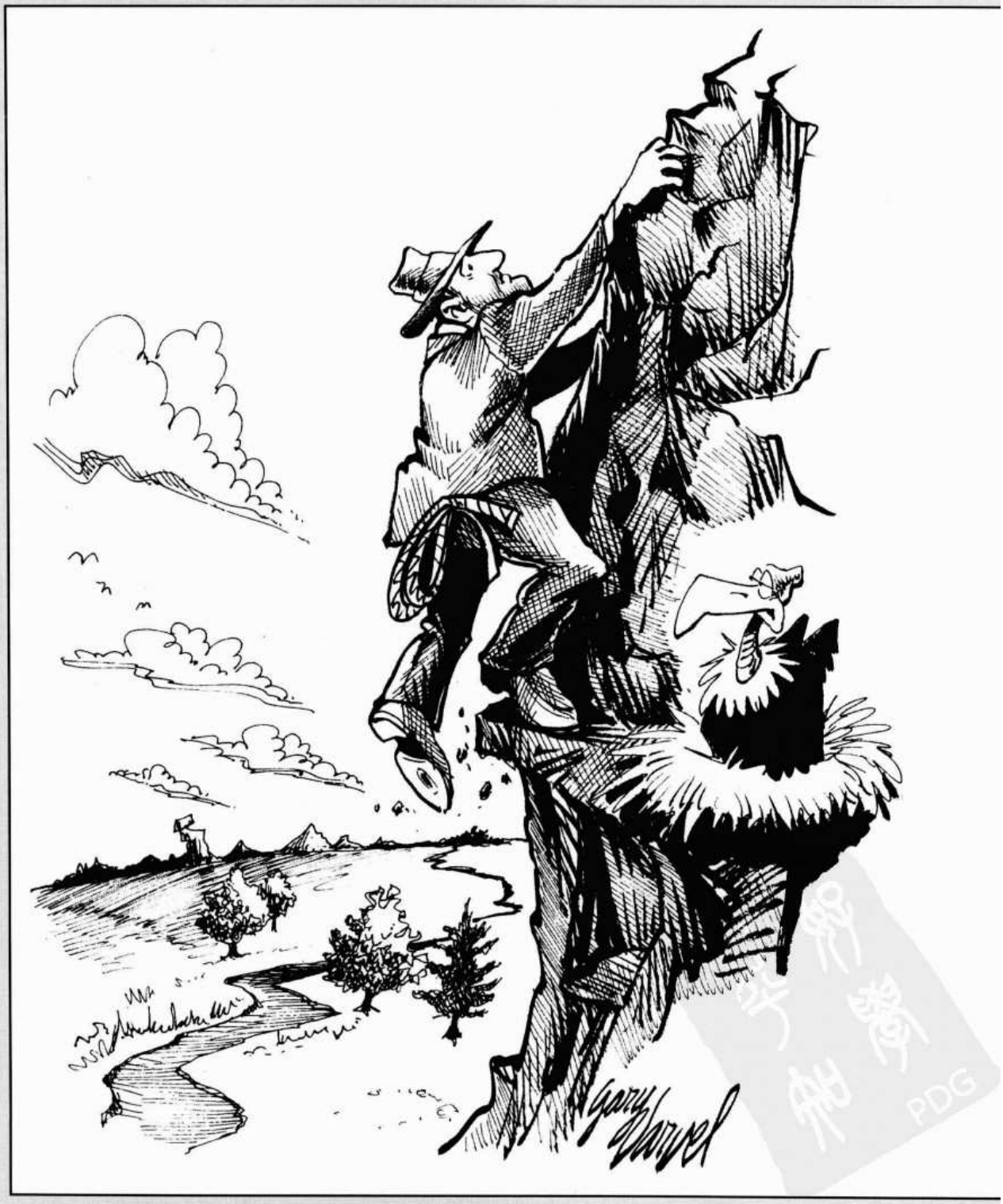
1. 代码范例

```
#include <math.h>
po2 = pow(2.0, 2.0);
printf("2 raised to the 2nd power is %.0f\n", po2);
s2 = sqrt(49);
printf("The square root of 49 is %.0f\n", s2);
si628 = sin(6.28);
printf("The sine of 6.28 is %f\n", si628);
r01 = rand();
printf("A random number from 1 to 32767 is %f\n", r01);
```

2. 代码分析

这些代码行给出了一个例子，讲解了如何使用本章描述的一些数学函数。程序包含了4对语句。每一对语句包含了一个函数调用和对应的结果输出。

请注意`rand()`是程序中唯一没有参数的函数。如果`rand()`基于你输入的一个值产生结果，那么`rand()`就不会是非常随机的了。



Part 4 第四部分

C 程序和大量数据

本部分内容

- 第 21 章　C 语言如何处理列表
- 第 22 章　搜索数据
- 第 23 章　排序
- 第 24 章　我的名字叫指针
- 第 25 章　数组和指针有什么不同
- 第 26 章　有效管理内存空间
- 第 27 章　结构体



21



val存放了被赋的值
179



在定义数组时，总是
要指定下标数目
177



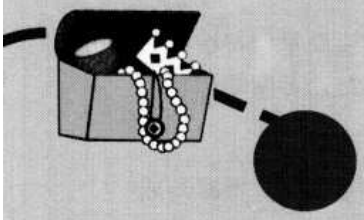
让你的数组足够大
以存放所有的值
176



所有C数组的下标从0开始
177



在初始化数组之前不要使用它
180



第21章

C语言如何处理列表

数组

本章内容

复习数组
176

数组赋值
179

有关数组的更多内容
180

关于本章的一个非常好的消息是完全没有讲述新的知识。在讲解把字符串存放在字符数组里时，你就与数组打过交道了。本章只是回顾数组的概念并阐明你可以创建任何数据类型的数组，而不仅仅是char数据类型。

如你所知，字符组成的数组不过是一个有名字的字符列表。类似地，整数组成的数组不过是一个有名字的整数列表，而浮点数组成的数组不过是一个有名字的浮点数列表。不需要用不同的名字来访问数组中的每一个元素，只需要用数组名加上带下标的方括号即可。

21.1　复习数组

所有数组包含的值都叫做元素（element）。一个数组只能包含相同类型的元素。换句话说，你不可能有一个包含浮点值、字符值和整型值的数组。

定义数组与定义普通的非数组变量几乎一样。定义普通变量时，你只需要在变量名前面指定它的数据类型：

```
int i;    /* Defines a non-array variable */
```

定义数组时，你必须在名字后面加上方括号（[]）并指定可能存放的最大元素个数：

```
int i[25];   /* Defines the array */
```

如果想用初始字符串来初始化一个字符数组，你可以这样写：

```
char name[6] = "Italy";  /* Leave room for the null! */
```



警告　一旦为数组定义了某个大小，就不要试图存储比原始大小更多的元素。在定义了上面的name之后，可以用strcpy()函数在name中存储比Italy更长的字符串，但是结果可能是灾难性的，因为内存中的其他数据会被意外地改写。如果另一变量正好定义在name后面，那么试图在name中存储太长的字符串会改写其他变量的数据。



线索　如果初始的数组需要比要赋的初值大，就在定义数组时指定一个更大的数组大小，像这样：

```
char name[80] = "Italy";  /* Leaves lots of
                             extra room */
```

这样做使你有空间存放比Italy长得多的字符串。例如，你可能想用gets()从用户处获取字符串，很容易出现比Italy更长的字符串。



警告 使数组足够大以存放足够多的值，但是不要过度。不要使数组比你真正需要的大很多。数组会占用大量的内存，而且你存储的元素越多，你的程序和其他变量能用的内存就越少。

在定义数组时，可以通过把数组的数据放在大括号里并在数组名后面跟上等号来初始化数组。例如，下面的语句既定义了一个整型数组又用5个值进行了初始化：


```
int vals[5] = {10, 40, 70, 90, 120};
```



请跳过这里，这有点儿难 图21-1说明了在定义之后，val在内存中的存储。方括号中的数字代表下标。在数组末尾没有null零，因为null零只作为存储在字符数组中字符串的结束符。

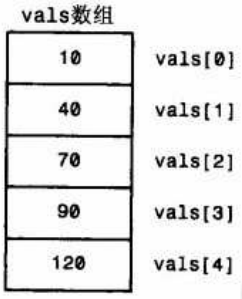


图21-1 定义和初始化vals数组之后



说明 所有C语言的数组下标都从0开始。

下面的语句定义并初始化了两个数组——一个单精度浮点数组和一个双精度浮点数组。因为C语言是自由格式的，你可以把初始化列表写在多行上，如annualSal数组那样。

```
float money[10] = {6.23, 2.45, 8.01, 2.97, 6.41};
```

```
double annualSal[6] = {43565.78, 75674.23, 90001.34,
                       10923.45, 39845.82};
```

你也可以定义字符数组，并用单个的字符来初始化：

```
char grades[5] = {'A', 'B', 'C', 'D', 'F'};
```

因为null零不是最后一个字符元素，所以虽然grades由单个字符组成，但是它不是字符串。如果最后一个元素被初始化为'\0'（代表null零），那么你就可以把grades看成字符串并用puts()或printf()和%s转化代码来打印它。下面的代码定义了在italCity数组中存放了一个字符串：

```
char italCity[7] = {'V', 'e', 'r', 'o', 'n', 'a', '\0'};
```

你不得不承认，用字符串来初始化这样的字符数组会更简单：

```
char italCity[7] = "Verona";    /* Automatic null zero */
```

我们应该回到数值的数组上，它才是本章的重点。在下面名为nums的数组末尾有null零吗？

```
int nums[4] = {5, 1, 3, 0};
```

在nums的末尾没有null零！要当心，nums不是字符数组，而且不能存放字符串。在数组末尾的0是一个普通的数值0。它的位模式（关于数据的内部表示的计算机术语）与null零很像。但是你不能把nums中存放的看成是字符，因为nums被定义成整型数值数组。



请跳过这里，这有点儿难　定义数组时，请一定要指定下标的数字！然而，这条规则有一个例外——如果在定义数组时赋初值或设定值，你可以让方括号为空：

```
int ages[5] = {5, 27, 40, 65, 92]};  /* Ceccect */
int ages[]; /* IncoPrect */
int ages[] = {5, 27, 40, 65, 92]};  /* Ceccect */
```



说明　sizeof()返回的是为数组预留（reserved）的字节数，而不是元素的个数。例如，如果一个浮点数在你的电脑上占用4个字节，那么有8个元素的浮点数组将总共占用32个字节的内存。如果在定义数组之后用sizeof()对其进行运算，将返回值32。



线索 如果想把数组的每一个元素都清零，你可以用C语言提供的快捷方式来实现：

```
float amount[100] = {0.0};  /* Zeroes-out all of
                               the array */
```

如果你没有初始化数组，C语言也不会帮你初始化。在给数组赋值之前，不知道数组中到底存放的是什么。这条规则的唯一例外是如果你在定义数组时初始化了至少一个数组元素，那么大多数C编译器会把所有的数组元素清零。前面线索部分的代码能起作用是因为在amout中第一个元素的位置存放了一个值，C语言于是把其他元素也用0填充了（即使第一个元素被初始化为123.45，C语言仍然会把其他元素用0填充）。

21.2 数组赋值

在定义数组时你不能总是知道数组的内容。通常，数组的值来自于磁盘文件、计算或者用户的输入。字符数组很容易用字符串来填充，因为C语言提供了strcpy()函数。你可以一次一个元素地填充其他类型的数组。没有像strcpy()这样快捷的函数，可以把很多整数或浮点数填充到数组中。

下面的代码定义了一个整型数组并要求用户输入存放在数组中的值。普通变量都有不同的名字，与之不同，数组元素容易操作因为你可以用循环对下标计数，如下所示：

```
int ages[3];
for (i = 0; i < 3; i++)
  { printf("What is the age of child #%d? ", i+1);
    scanf(" %d", &ages[i]);  /* Gets next age from user */
  }
```



线索 21点游戏程序（参见附录B）中initCardScreen()函数使用条件运算符来产生代表不同纸牌的不同数值：

```
val = (val == 14) ? 1 : val; /* If val is 14,
                                reset to 1 */
```

程序中val不断递增。然而，当val变成14时，这个条件语句就把val重置为1。值1到13代表了每种花色的13张纸牌（从A到K）。4组从1到13的值填充了纸牌数组，而val存放被赋的值。

21.3　有关数组的更多内容

本章讲解了数组的作用。下一章讲解如何按顺序排序数组以及如何在数组中搜索特定的值。



奖励

- 用数组来存放相同数据类型的值的列表。
- 用下标来访问数组的单个元素。
- 如果想遍历数组的每一个元素（初始化、打印或修改它），可以用for循环来实现。



陷阱

- 在数组中，不要使用比所定义的更多的元素。
- 在初始化数组之前，不要使用它。

21.4　小结

本章讲解了如何在叫做数组（array）的列表中存储数据。数组不过是一堆变量。每个变量都有相同的名字（数组名）。用数值的下标（subscript）来区分数组中的变量（数组元素）。第一个数组元素的下标是0，其余的元素下标依次增加。

数组由名字后面的方括号来刻画。当你想访问一个数组元素时，把数组下标写在方括号中即可。

1. 代码范例

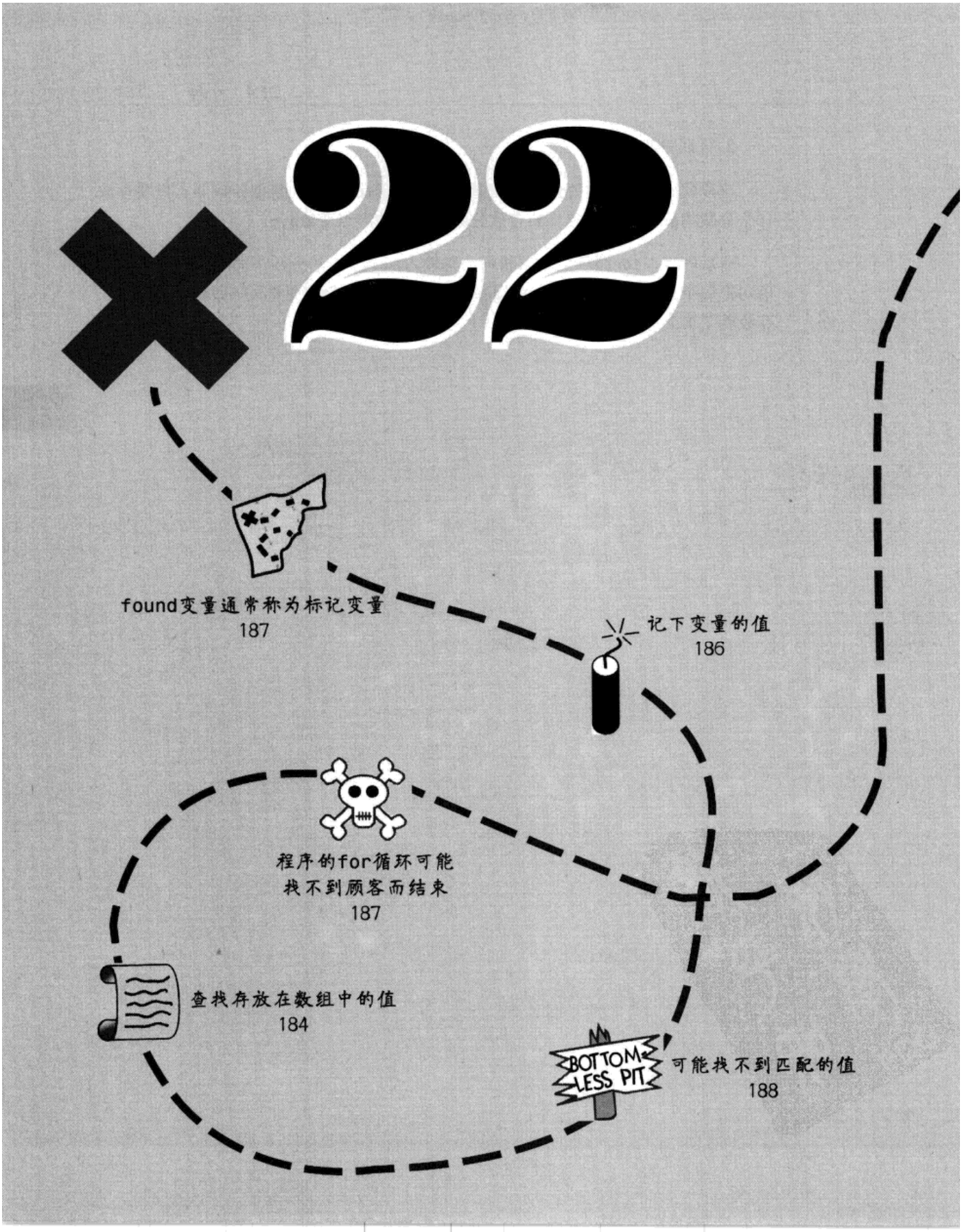
```
int i;
float grades[10];  /* Creates the array */
float avg = 0;
for (i = 0; i < 10; i++)
  { printf("What is student number %d's grade? ", i+1);
    scanf(" %f", &grades[i]);  /* Gets each grade
                                  from teacher */
    avg += grades[i];  /* Adds to total scores */
  }
  avg /= 10;  /* Computes average */
  printf("\nThe average of all grades is %.2f.\n", avg);
```

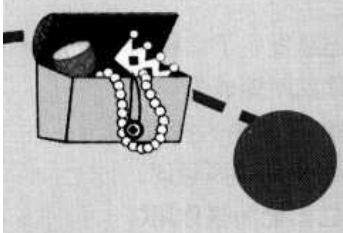
2. 代码分析

这段代码首先创建要使用的变量，包括用于for循环的整型计数器、将要存放10个分数的grades数组和将要存放分数平均值的浮点变量avg。

然后程序在for循环中要求教师分别输入10个学生的分数。循环中的最后一条语句把每个分数都加到一起。在程序的最后，存放总分的变量（叫做avg）被10除，存放着平均分数。

22


found变量通常称为标记变量
187
记下变量的值
186
程序的for循环可能
找不到顾客而结束
187
查找存放在数组中的值
184
BOTTOM
LESS PIT
可能找不到匹配的值
188



第22章

搜索数据

遍历数组

本章内容

填充数组
184

搜索
185

你买了这本书是希望尽可能不再痛苦地学习C语言——而这已经发生了。你知道有些事发生了，不是吗？然而，如果不掌握一些搜索和排序，你就不能成为一流的程序员。有许多书介绍了关于搜索和排序的技术，下面两章只讲解最简单的技术。然而，在开始学习之前，作为预先的警告，本章和下一章可能会引出你许多疑问。如果你喜欢搜索和排序，可以查看附录A，那里列出了许多书，它们能加强你用C语言进行搜索和排序的能力。

你将发现本章和下一章与其他章有一点不同。这两章不是教你新的C语言特性，而是阐明你已经学过的C语言元素的使用，且重点是数组。你将看到在第21章中学过的数组概念的程序。对数组有了更好的理解之后，第25章就讲解一个有时比数组更便于使用的另一种方式。

22.1　填充数组

在第21章中提过，你的程序将用几种方式来填充数组。一些数组（比如12个月份的天数、历史的温度和最后一年的销售记录）都能预先知道。你可能在定义时或者在定义之后用赋值语句来初始化这些数组。

你也将会使用用户输入的数值来填充数组。只有当客户发出订单时，顾客—订单程序才会获取数据。科学的实验室只有在科学家收集了结果后才知道测试值。

其他数据可能来自于磁盘文件。顾客记录、存货值和学生成绩单如果让用户在程序每次运行时输入，就太长了。

现实中，你的程序会结合这3种方法来填充数组：

- 赋值；
- 用户数据输入；
- 磁盘文件。

本书中，程序要保持简单。在学习磁盘文件之前，你将会看到用赋值语句来填充数组，还有简单的用户输入（你就是用户）。



说明　在填充了数组之后，最重要的是查找存放在数组中的值。

22.2 搜索

设想下面的场景：你的程序包含一个存放顾客ID号的数组和一个存放对应顾客欠款额的数组。这样的数组通常称为*并行数组*（parallel array），因为它们是同步的。也就是说，顾客ID数组的14号元素包含了顾客编号，他欠下了欠款额数组的14号元素中的欠款额。

趣闻轶事
搜索的值也叫做键值。

程序可能在第一次启动时用磁盘数据填充了这两个数组。当某个顾客发出新的订单时，程序的任务是找出顾客的欠款额并且在顾客已经欠款超过100美元时停止订单。

简而言之，程序的任务如下：

- 获取顾客的ID号（键值）。
- 在顾客数组中搜索与键值匹配的ID号，从而得到顾客的索引。
- 用上一步找到的索引在欠款数组中查找顾客的欠款。
- 如果顾客欠款已经超过100美元，就通知顾客。

下面的程序完成这些任务。实际上，程序只维护了一张包含10个顾客的列表，因为你还没学会处理磁盘输入（很快就会学习了）。程序在数组定义时对其初始化。只维护10个数组元素对（顾客ID数组和相应的欠款额数组）使得数组的定义比较简单。



说明 就像你将在下一章中看到的排序程序那样，这个程序比本书前面的其他程序都更长一些。但是你已经做好了阅读较长程序的准备，因为你有一个像我这样的好老师！

在输入这个程序并运行它之前，最好先好好地研究它。看看你是否能从代码和注释中获取程序的要点。在代码之后会有详细的讲解。

趣闻轶事
这个程序使用了一种顺序的搜索技术。

```
/* Searches for balances */
#include <stdio.h>
main()
{
  int ctr;  /* Loop counter */
  int idSearch;  /* Customer to look for (the key) */
  int found = 0; /* 1 (true) if customer is found */

  /* Defines the ten elements in each of the parallel arrays */
```

```
int custID[10] = {313, 453, 502, 101, 892,
                  475, 792, 912, 343, 633 };
float custBal[10] = {  0.00,  45.43,  71.23, 301.56,  9.08,
                     192.41, 389.00, 229.67,  18.31, 59.54};

/* Interact with the user looking for a balance */
printf("\n\n*** Customer Balance Lookup ***\n");
printf("What is the customer number? ");
scanf(" %d", &idSearch);

/* Now, look for the balance in the arrays */
for (ctr=0; ctr<10; ctr++)
{  if (idSearch == custID[ctr])    /* Found? */
   {  found = 1;       /* Yes, found is true */
      break;           /* No more search needed */
   }
} /* Not found yet, loop again to try another match */

/* If the for-loop finishes, the customer ID was
   found or all customers were searched unsuccessfully  */
if (found)           /* Cleaner than if (found == 1) */
  { if (custBal[ctr] > 100.00)
      { printf("\n** That customer's balance is $%.2f.\n",
               custBal[ctr]);
        printf("   No credit!\n");
      }
    else    /* Balance is less than 100.00 */
     { printf("\n**The customer's credit is good!\n");
     }
  }
 else   /* Not found */
   { printf("** You must have typed an incorrect customer
             ID.");
     printf("\n   ID %3d was not found in list.\n", idSearch);
   }
  return 0;
}
```



请跳过这里，这有点儿难　是的，这个程序比较长。然而，本书的大多数程序都是短小的代码示例。仔细阅读这个程序并弄懂它在做什么对你来说是一个很好的练习。用铅笔和纸记下变量的值。

程序有3种可能：

- 顾客的欠款少于100美元；
- 顾客的欠款太高（大于100美元）；
- 顾客的ID不在列表中。

下面是程序的3次运行，分别表明了上面3种情况：

```
*** Customer Balance Lookup ***
What is the customer number? 343

**The customer's credit is good!


*** Customer Balance Lookup ***
What is the customer number? 583
** You must have typed an incorrect customer ID.
   ID 583 was not found in list.
*** Customer Balance Lookup ***
What is the customer number? 101

** That customer's balance is $301.56.
   No credit!
```


程序的第一部分定义了两个数组，并用顾客ID号和匹配的欠款额来初始化它们。如你所知，在最初定义数组时，可以用赋值运算符（=）为数组赋值。

在打印出输入提示并要求用户输入一个顾客ID号之后，程序用一个for循环遍历并行数组来寻找用户输入的顾客ID。如果找到了，就把变量found置为真（1）以供后面使用；否则，found仍然为假（0）。



线索 变量found通常称为标记（flag）变量，因为它为程序的其他部分做了标记，表明顾客ID是否找到。



警告 程序的for循环可能找不到顾客而结束。for循环后面的代码无法知道是for的break语句触发了循环的提前结束（意味着找到了顾客）还是自然结束。因此，变量found使得for循环后面的代码知道for循环是否找到了顾客。

一旦for循环结束，顾客就找到（或者没找到）。如果找到了，可能出现下面两种情况：

- 欠款已经太高；
- 欠款还可以接受。

无论是哪种情况，都会告诉用户结果。如果顾客没有找到，程序就会通知用户并结束。

现实生活中的程序是怎样的呢？太难说清了？仔细地多看几遍程序。你会明白程序就是按照人工查找顾客列表相同的步骤执行的。



线索　真正重要的是如果有成千上万的顾客，而数组是用磁盘文件来初始化的，相同的搜索代码也能工作！数据量不影响程序的逻辑（只影响数组初始化的方式）。



线索　前面程序的搜索是顺序搜索（sequential search），因为顾客ID数组是从头到尾搜索直到找到匹配的值。当你的编程技巧提高后，还将学习更高级的搜索方法。下一章中，你将看到数组排序是怎样加快数组搜索速度的。还有一些高级的搜索技术叫做二分搜索（binary search）和斐波那契搜索（Fibonacci search）。



奖励

- 填充数组只是第一步，一旦填充之后，程序就应该操作数据了。
- 在学习更多的搜索方法之前，请使用顺序搜索，因为这是要首先掌握的最简单的搜索技术。



陷阱

- 别忘了可能找不到一个匹配的值。始终要假设你要搜索的值不在列表中，并且始终要编写搜索不到匹配的值的代码。

22.3　小结

本章讲解了如何在数组中寻找某个值。你学习了如何基于键（key）寻找数组的值。键是用户输入的一个值。你将会经常像本章这样搜索并行数组。一个数组（关键数组）存放要搜索的值。如果搜索成功了，另一个数组提供了所需的数据并且可以把结果报告给用户。如果搜索没有成功，你也需要让用户知道。

1. 代码范例

```
/* Searches for letters */
#include <stdio.h>
#include <stdlib.h>
main()
{
```

```
  char letters[50];
  char userInit;
  int ctr, nl, numInits=0;
  /* Store random letters from A to Z */
  for (ctr=0; ctr<50; ctr++)
    { letters[ctr] = 65 + (rand() % 26); }
  /* Get the user's search initial */
  printf("What is your first initial? ");
  userInit = getchar();
  nl = getchar();  /* To discard the newline */
  /* Find the number of occurrences */
  for (ctr=0; ctr<50; ctr++)
    { if (letters[ctr] == userInit)
        { numInits++; }
    }
  printf("\nIn the array, there were %d of your initials.\n",
         numInits);
  return 0;
}
```


2. 代码分析

这个程序首先用rand()函数把字母表中的字母存放在50个字符数组的元素中。公式rand()%26产生从0到25的数字。这些数字加上65得到从65到90的ASCII表值，这些值与从A到Z的大写字母匹配。

一旦50个随机字母存放在letters数组中，程序就要求用户输入他的初值。然后用一个for循环在数组中寻找匹配的值。每当找到一个匹配的值，就给变量numInits加1。一旦程序搜索完全部的数组元素，就会用一条printf()打印出总的匹配数。

23



rand()在不同的编译器
上产生不同的结果
195



按降序对列表排序
195



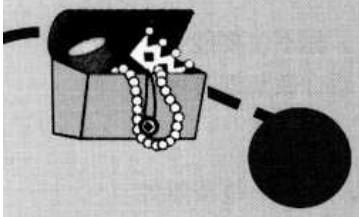
保持数组排好序并
不总是十分容易
199



按字母顺序排列
195



先给数组排序
199



第23章

排　　序

冒泡排序

本章内容

排序
192

加快搜索
196

排序（sorting）是一个计算机术语，也就是调整列表的顺序。你不仅要能够在数组中找到数据，还经常需要按某种顺序给数组排序。电脑非常适合做数据排序的工作，而数组可以用来存储排序后的数据。

你的程序不会总是按你希望的顺序存放数组的数据。例如，学生不是按照姓氏的字母顺序招收的，即使大多数学校按照这样的顺序打印学生列表。因此，一旦收集了学生数据，学校的计算机程序必须用某种方式把数据按照姓氏的字母顺序排列。

本章讲解最简单的计算机排序技术，叫做冒泡排序（bubble sort）。

23.1　排序

如果想让一个字母或名字的列表按照字母顺序排列，或者让一个销售值列表按照升序（ascending）（升序意味着从低到高，而降序意味着从高到低）排列，你就要用到排序程序。当然，需要排序的列表存储在数组中，因为数组的值很容易通过下标来重新安排。

想一想如果把一副牌扔到空中并散落到地上，你要怎样把它们按顺序收起来。你应该一张张地捡起来，看看当前的牌与手中其他牌的关系。你经常需要重新安排已经在手中的牌。在对数组排序时，也会用到类似的处理办法。你经常不得不重新安排数组中的值。

有几种计算机排序方法。本章将教你冒泡排序（bubble sort）。冒泡排序与其他排序方法相比不是非常高效，但却是最易于理解的。名字冒泡排序来源于此排序的特性。在排序的过程中，每一遍扫描都使较小的值“浮”到列表的上端。图23-1说明了用冒泡排序对5个数进行排序的过程。

下面的程序对10个数字的列表进行排序。数字是用`rand()`函数随机生成的。冒泡排序只不过是嵌套的`for`循环。内层循环扫描列表，并把不按任何符合顺序的一对数字进行交换。外层循环控制内层循环执行次数（为列表中的每个元素执行一次）。

趣闻轶事
其他排序方法有快速排序、堆排序和Shell-Metzner排序。

很多改进后的冒泡排序程序会测试在每一次内层循环迭代时，是否有交换发生。如果没有发生交换，外层循环就提前结束（通过`break`语句）。因此，如果在循环开始时列表已经排好序了，或者如果只需要扫描几遍就能完成排序，那么外层循环就可以提前结束。

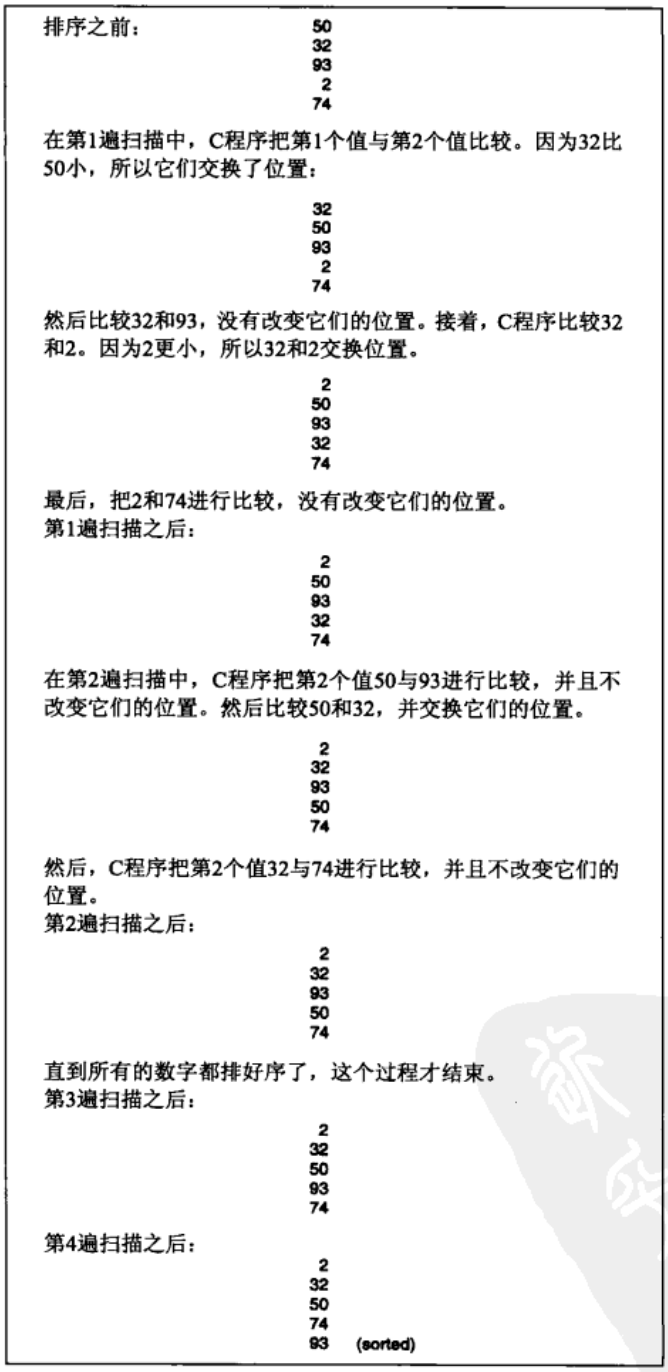


图23-1 在每一遍扫描的过程中，较小的值“浮”到数组的顶部

```
#include <stdio.h>
#include <stdlib.h>
/* Program that sorts a list of 10 numbers */
main()
{
  int ctr, inner, outer, didSwap, temp;
  int nums[10];  /* Will hold the 10 numbers */

  /* Fills array with random numbers from 1 to 100 */
  for (ctr = 0; ctr < 10; ctr++)
    { nums[ctr] = (rand() % 99) + 1; }

  /* Prints the list before it is sorted */
  puts("\nHere is the list before the sort:");
  for (ctr = 0; ctr < 10; ctr++)
    { printf("%d\n", nums[ctr]); }

  /* Sorts the array */
  for (outer = 0; outer < 9; outer++)
    { didSwap = 0;  /* Becomes 1 (true) if list
                       is not yet ordered */
      for (inner = outer; inner < 10; inner++)
        {  if (nums[inner] < nums[outer])
             {  temp = nums[inner];
                nums[inner] = nums[outer];
                nums[outer] = temp;
                didSwap = 1;  /* True because a swap
                                 took place */
             }
        }
      if (didSwap == 0)  /* Quits if list is now sorted */
        { break; }
    }
/* Prints the list after it is sorted */
  printf("\nHere is the list after the sort:\n");
  for (ctr = 0; ctr < 10; ctr++)
    { printf("%d\n", nums[ctr]); }
  return 0;
}
```

这个排序程序的输出如下：

```
Here is the list before the sort:
71
54
58
29
31
78
2
77
82
71
```

```
Here is the list after the sort:
2
29
31
54
58
71
71
77
78
82
```



线索 你的输出可能与上面例子的输出不同，因为rand()函数在不同的编译器上会产生不同的结果。不过，关键是查看程序产生的10个随机数的放置顺序，只要在程序执行完成后它们被排好序就可以了。

下面是内层循环中交换两个变量的语句：

```
temp = nums[inner];
nums[inner] = nums[outer];
nums[outer] = temp;
```


换句话说，如果一次交换需要发生（第1个数比第2个数大），程序就必须交换nums[inner]和nums[outer]。

你可能想知道为什么在交换两个变量时要用到另一个变量temp。我们很容易像下面这样交换两个变量（不正确的）：

```
nums[inner] = nums[outer];  /* Does NOT swap the */
nums[outer] = nums[inner];  /* two values        */
```

第1条赋值语句把nums[inner]的值改变了，于是第2条赋值语句就无值可赋了。因此，交换任何两个变量时都需要第3个变量。



请跳过这里，这有点儿难 如果你想把列表按降序排列，只需要把在交换变量的代码之前的小于号（<）改成大于号（>）即可：

```
/* Top of program would go here */
{  if(num[inner] > nums[outer])
   /* Rest of program follows */
```



说明 如果想把一个字母表按字母顺序排列，你可以像这样通过测试并交换字符数组的值来实现。在第25章中，你将学习如何存储可以排序的字符串数据的列表。

23.2　加快搜索

有时候，对数据的排序可以加快数据的搜索。在上一章中，编写了一个程序，它搜索一个顾客ID数组以寻找匹配的用户的值。如果找到了一个匹配，就用相对应的顾客欠款额（在另一个数组中）来进行信用检查。顾客的ID 值没有按任何顺序存储。

也可能找不到用户输入的顾客ID。可能用户输入的ID号不正确，或者这个顾客ID号没有存储在数组中。程序员必须在搜索了每一个数组元素之后，才能够意识到无法找到这个顾客ID号。

然而，如果在搜索开始之前数组就按照顾客ID的顺序存储，程序就不必在查看了每一个数组元素之后才能判断出找不到一个匹配的顾客ID。如果顾客ID数组被排好序了，并且一开始就没有匹配的顾客ID，那么程序马上就知道不可能再找到匹配的值了。看一看下面未排序的顾客ID列表：

```
313
532
178
902
422
562
```

假设程序要搜索的顾客ID是413。当使用未排序的数组时，程序不得不把413和每一个元素进行对比。

如果数组中包含几百或几千个值而不是只有6个值，那么电脑就需要更长的时间才能发现找不到匹配的值，因为每一次搜索都要求查看每一个元素。然而，如果数组中的值已经排好序了，程序不必扫描整个列表就能发现不可能找到匹配的值。下面是相同的值的列表，从小到大排列：

```
178
313
422
532
562
902
```

排序后的列表加快了搜索的速度。现在如果你要搜索的顾客ID是413，程序在查看了3个数组的值之后就会停止搜索。第3个元素是422，由于422大于413，你的程序就可以停止搜索了。因为422应该跟在413后面，所以可以停止搜索了。



说明 在极端的情况下，在已排序的数组中搜索并不一定比在未排序的数组中搜索更快。例如，如果你在前一个列表中搜索顾客ID 998，那么你的程序就得搜索了所有6个值才能发现998不在列表中。

下面的程序结合了上一章中介绍的顾客ID搜索程序和本意介绍的排序程序。顾客ID值被排好序，然后要求用户输入要搜索的顾客ID。程序将判断顾客的欠款是否少于100美元。然而，如果此顾客ID不在列表中，程序会提前结束搜索。也许因为只有10个数组元素会让这个程序看起来有点夸张，但是如果有1000个顾客，那么代码将会大大不同。

```
/* Searches a sorted list for balances */
#include <stdio.h>
main()
{
  int ctr;  /* Loop counter */
  int idSearch;  /* Customer to look for (the key) */
  int found = 0; /* 1 (true) if customer is found */

  /* Defines the ten elements in each of the parallel arrays */
  int custID[10] = {313, 453, 502, 101, 892,
                    475, 792, 912, 343, 633 };
  float custBal[10] = {  0.00,  45.43,  71.23, 301.56,  9.08,
                       192.41, 389.00, 229.67,  18.31, 59.54};
  int tempID, inner, outer, didSwap;   /* For sorting */
  float tempBal;                       /* and swapping */

  /* First, sort the arrays by customer ID */
    for (outer = 0; outer < 9; outer++)
      {  didSwap = 0;  /* Becomes 1 (true) if list
                          is not yet ordered */
         for (inner = outer; inner < 10; inner++)
           {  if (custID[inner] < custID[outer])
              {  tempID = custID[inner];     /* Must swap    */
                 tempBal = custBal[inner];  /* BOTH arrays! */
                 custID[inner] = custID[outer];
                 custBal[inner] = custBal[outer];
                 custID[outer] = tempID;
                 custBal[outer] = tempBal;
                 didSwap = 1;  /* True because a swap
                                  took place */
              }
           }
         if (didSwap == 0)  /* Quits if list is now sorted */
            { break; }
      }
  /* Interact with the user looking for a balance */
  printf("\n\n*** Customer Balance Lookup ***\n");
```

```
    printf("What is the customer number? ");
    scanf(" %d", &idSearch);

    /* Now, look for the balance in the arrays */
    for (ctr=0; ctr<10; ctr++)   /* Search maximum of 10 values */
    {  if (idSearch == custID[ctr])    /* Found? */
         {  found = 1;       /* Yes, found is true */
            break;           /* No more search needed */
         }
       if (custID[ctr] > idSearch)   /* Don't go too far */
         { break; }  /* Know early the ID isn't in the list */
    } /* Not found yet, loop again to try another match */

    /* If the for-loop finishes, the customer ID was
       found or all customers were searched unsuccessfully  */

    if (found)          /* Cleaner than if (found == 1) */
      { if (custBal[ctr] > 100.00)
          { printf("\n** That customer's balance is $%.2f.\n",
                   custBal[ctr]);
            printf("   No credit!\n");
          }
        else    /* Balance is less than 100.00 */
         { printf("\n**The customer's credit is good!\n");
         }
      }
     else   /* Not found */
       { printf("** You must have typed an incorrect customer
ID.");
         printf("\n   ID %3d was not found in list.\n", idSearch);
       }
     return 0;
}
```



说明　除了附录B中的21点游戏，前面的程序是本书中最难理解的程序。掌握了这个程序，你就不再是初学者的水平了。一旦你掌握了这里的逻辑，就要祝贺你并向你致敬。看，用C语言编程根本不难！

在看这个程序之前，你掌握了数组搜索和数组排序。这个程序只是把这两个过程合在了一起。这个程序所做的唯一的额外的工作是在搜索中使两个并行数组保持同步。正如在搜索代码的主体中看到的那样，当顾客ID的元素被交换（在custID数组中）时，在顾客欠款额数组中对应的（通过下标）元素也被交换了。

搜索可能因为下面的语句而提前结束：

```
if (custID[ctr] > idSearch)   /* Don't go too far */
  { break; }  /* Know early the ID isn't in the list */
```

当有几千个数组元素时，这个if语句会节省很多处理时间。



警告 保持数组的排序状态并不总是容易的和高效的。例如，你不想让程序每次在添加、修改或删除一个值时，都对庞大的数组进行排序。在把几千个值存储到数组中之后，对数组进行排序会占用很多时间，即使对于快速的计算机也一样。有高级的方法来操作数组，使得总是按顺序插入元素。然而，这样的技术超出了本书讨论的范围。在这里不要把事情变得太复杂了。



奖励

当你想从低到高排列数组的值时，就使用升序排序。
当你想从高到低排列数组的值时，就使用降序排序。
嵌套的for循环，比如在本章中见到的，是实现冒泡排序的完美语句。



陷阱

不要交换两个变量的值，除非你引入第3个工作变量来存放中间值。
排序程序不一定很难。从本章给出的程序开始学习，并根据需要修改它。
别忘了先给数组排序。这样可以加快搜索值的速度。

23.3 小结

本章讲解了冒泡排序方法。你不需要学习任何新的C命令来实现排序。排序最能体现出数组的优势。它表明了数组比起单独命名的变量来说是更好的存储方法。数组的下标可以用来遍历数组并交换其中的值，当需要时，对其排序。

1. 代码范例

```
/* Sorts the array backwards (in descending order) */
for (outer = 0; outer < 9; outer++)
  { didSwap = 0;  /* Becomes 1 (true) if list
                     is not yet ordered */
for (inner = outer; inner < 10; inner++)
  {  if (nums[inner] > nums[outer])  /* Descending
                                        test */
     { temp = nums[inner];  /* Swaps if needed */
       nums[inner] = nums[outer];
       nums[outer] = temp;
       didSwap = 1;   /* True because a swap
```

```
                                took place */
        }
    }
    if (didSwap == 0)  /* Quits if list is now sorted */
      { break; }
}
```

2. 代码分析

这段代码构成了完整的降序排序代码，用来对本章的10个随机数的列表进行排序。升序排序和降序排序的唯一不同是在交换代码上面的if比较部分。

这条if语句标有注释/* Descending test */。嵌套的循环测试列表中的数字对。如果数字对中最下面的值比第2个值更大，那么就执行交换代码以交换这两个值。交换代码由4行构成，标有注释/* Swaps if needed */。

×24



对变量名的支持
206



乘号和解引用符号
之间的不同
205



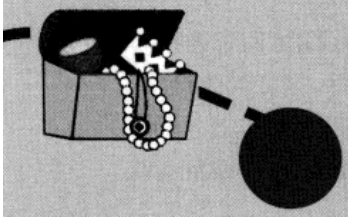
段和偏移量
205



指针变量的名字
206



无需知道C语言为变量
分配的准确地址
208



第 24 章

我的名字叫指针

使用指针

本章内容

内存地址
204

定义指针变量
205

使用取值运算符*
207

指针变量（pointer variable），通常称为指针（pointer），使得你用C语言可以做比用其他不支持指针的编程语言能做更多的事。第一次学习指针时，你可能会问“什么是指向？”即使在你掌握了指针之后，你可能还会问同样的问题！事实上，指针为C编程提供了真正强大的手段。不过，本书只介绍了很少的指针内容。你在这里学到的概念将为以后的C编程打下基础。

24.1　内存地址

趣闻轶事
BASIC、FORTRAN和COBOL语言不支持指针。

电脑中有大量的内存。内存中存储着程序，也存储着程序的变量。正如每一间房子都有不同的地址一样，每个内存位置也有不同的地址。像房子的地址一样，内存地址都是唯一的。没有两个相同的地址。

图24-1表明了电脑中的内存。地址与存储在该地址上的数据没有任何关系。所有的电脑地址从0开始，并逐渐增加。

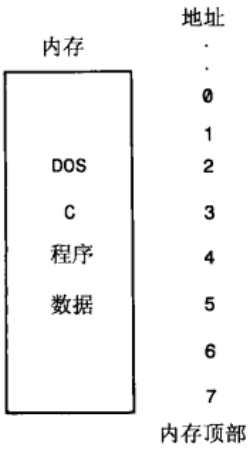


图24-1　电脑包含了很多内存，每个内存位置都有唯一的地址

如图24-1所示，内存像一个大的硬件数组一样工作，每个地址就是不同的下标，而每个内存位置就是不同的数组元素。

定义变量时，C语言会在内存中寻找未使用的位置并把名字与这个内存位置相关联。这是件好事。你只需要记住名字`orderNum`（假设你在定义变量时起名为`orderNum`），而不用记住有个数存储在内存地址34532上。名字`orderNum`比一个数字更容易记忆。



线索 对变量名的支持只是编译器提供的许多编程捷径中的一种。



警告 图24-1有一点易使人误解。PC的地址通常划分为段（segment）和偏移量（offset）。PC的内存被划分为代表内存的不同部分的段，就像书的章代表了书的不同部分一样。偏移量用来加到段的地址上，从而得到特定的内存位置。以后你们可能会在平滑内存模式（flat memory model）的计算机上编程，在这种模式下就不再需要用到段，给每个内存位置分配一个从0开始的顺序地址直到内存用尽为止[①]。

24.2 定义指针变量

与其他类型的变量一样，使用指针变量之前必须定义它。在深入学习之前，你需要学习两个新的运算符。表24-1列出了这两个运算符及其描述。

表24-1 指针运算符

运算符	描　述
&	取址运算符
*	取值运算符



请跳过这里，这有点儿难 你已经在前面见过*。C语言如何区分乘法和取值？你使用它们的上下文决定了C语言如何理解它们。你已经在scanf()中的变量前面见过&。scanf()中的&就是取址运算符。scanf()要求你传入非数组变量的地址。

下面的语句展示了如何定义整型和浮点型变量：

```
int num;
float value;
```

要定义整型指针变量和浮点型指针变量，只要插入一个*：

```
int * pNum;  /* Defines two pointer variables */
float * pValue;
```

① 现在的操作系统已经实现了这一点。——译者注



说明　指针变量的名字没有什么特别的。许多程序员喜欢让指针变量名以p开头，像这里这样，你也可以按自己的方式命名它们。p只是提醒你它们是指针变量而不是普通变量。



线索　所有的数据类型都有相应的指针数据类型。有字符型指针、长整型指针等。

指针变量存放了其他变量的地址。这是它们的主要用途。可以用取址运算符&把一个变量的地址赋给一个指针。在把一个变量的地址赋给一个指针之前，指针未被初始化，而且你不能用它们做任何事。

下面的代码定义了名为age的整型变量并在age中存储19。然后定义了名为pAge的指针并使它指向age。第2行告诉C程序把age的地址放到pAge中。

```
int age = 19;        /* Stores a 19 in age */
int * pAge = &age;   /* Links up the pointer */
```

你不能准确地知道C程序把age存储在什么地址上。然而，不管是什么地址，都存放在pAge中。当指针变量存放了另一个变量的地址时，事实上也就是它指向（point）那个变量。假设age存储在地址18826上（只有C程序准确地知道它存储在哪里），图24-2说明了内存的样子。

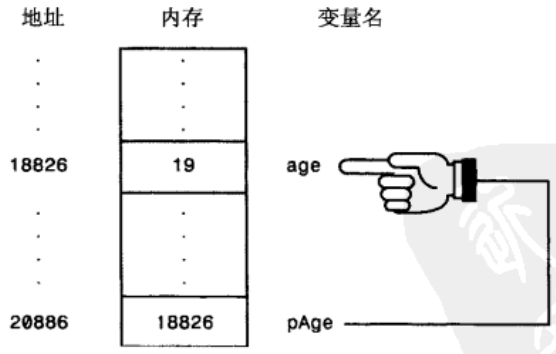


图24-2　如果pAge存放了age的地址，那么变量pAge就指向age



说明　你连续地定义两个变量并不意味着C语言会把它们连续地存储在内存中。C语言可能把它们存储在一起，也可能分开。



警告 绝对不要把一种类型的变量的地址赋给不同类型的指针变量。C语言只允许把一种类型的变量地址赋给用相同数据类型定义的指针。

*不是指针变量名的一部分。你将用*取值运算符（dereferencing operator）做好几件事，但是在指针的定义中，*的存在告诉C语言这个变量是一个指针而不是普通变量。下面的4条语句做了与前面两条语句完全相同的事情。请注意在把一个变量的地址赋给指针变量时不用*，除非你在同时定义了这个指针。

```
int age;      /* Defines a regular integer */
int * pAge;   /* Defines a pointer to an integer */
age = 19;     /* Stores 19 in age */
pAge = &age;  /* Links up the pointer */
```

24.3 使用取值运算符*

一旦把一个指针指向另一个变量后，你就可以通过对指针取值（dereferencing）来处理另一个值。取值意味着用指针来获取另一个变量。当你取值时，用*取值运算符。

简而言之，有两种修改age值的方法（假设变量像前面描述的那样被定义）：

```
age = 25;
```

和

```
*pAge = 25;  /* Stores 25 where pAge points */
```

请注意你可以用变量名或者对指向该变量的指针取值来进行赋值。你也可以用相同的方法来使用变量的值。下面是打印age值的两种方法：

```
printf("The age is %d.\n", age);
```

和

```
printf("The age is %d.\n", *pAge);
```

当函数处理传入的指针变量时，要使用取值运算符。在第32章中，你将学习如何把指针传给函数。当一个函数使用从另一个函数传来的指针变量时，你必须在变量名前使用取值运算符。附录B中的21点游戏程序在dealCard()函数中就是这么做的。下面的代码来自这个函数，说明了如何使用取值运算符：

```
subDraw = (rand() % (*numCards));  /* From 0 to numcards */
```

```
cardDrawn = cards[subDraw];
cards[subDraw] = cards[*numCards - 1];  /* Puts top
                                           card in */
```



奖励

- 要习惯内存的地址，因为它们构成了指针用法的基础。
- 用&来获取变量的地址。
- 用*来定义指针变量和对指针变量取值。只要你已经让pAge指向age，*pAge和age就引用了相同的内存位置。



陷阱

- 不要尝试把一种数据类型的指针变量指向另一种数据类型的变量。
- 无需知道C语言为变量分配的准确地址。你只需使用&，C语言将会帮你处理其余的事。
- 不要忘记在对指针取值时使用*，否则你将得到错误的值。
- 不要想一下子就掌握指针的精髓。只有在用C编程一段时间之后，你才会完全地欣赏指针。此时，指针看起来没有任何帮助。你可能觉得唯一的好处是知道了scanf()中&的真正意思。

24.4　小结

本章介绍了指针变量。指针变量只是一个存放着另一个变量的内存位置的变量。你可用变量名或对指针取值来引用被指向的变量。

指针在C语言中有很多用处，尤其是在高级的C编程中。在下一章你将要学到，数组只不过是一种特殊类型的指针。因为指针比数组提供了更好的灵活性，许多C程序员在掌握了指针之后就不用数组了。

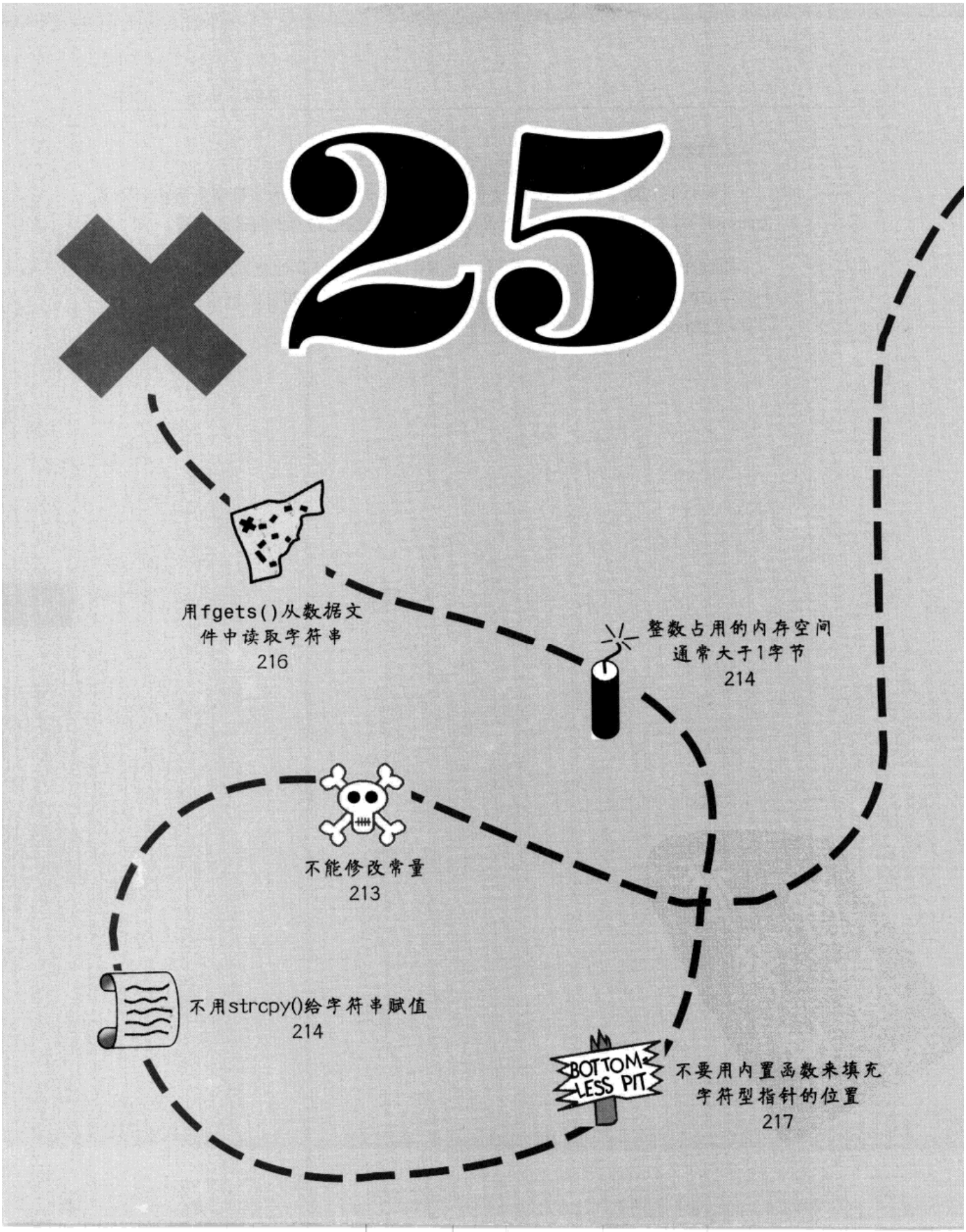
1. 代码范例

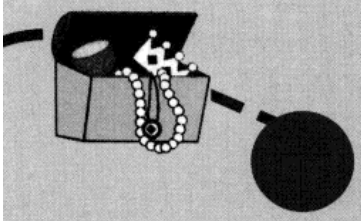
```
char initial = 'G';  /* Defines a character variable */
char * pInitial;  /* Defines a character pointer variable */
float score = 97.4;  /* Defines a floating-point variable */
float * pScore;  /* Defines a floating-point pointer variable */
pInitial = & initial;  /* Links the pointer to the data */
pScore = &score;
printf("The initial is %c.\n", *pInitial);
*pScore = 85.0;  /* Changes score, not pScore! */
```

2. 代码分析

这段代码初始了两个非指针变量——一个名为initial的字符型变量和一个名为score的浮点型变量。代码还定义了两个指针变量把它们指向普通变量。

如你在最后两句中所见，你可以用指针变量来获取其他变量的值。printf()不打印pInitial的值，而是打印了initial的值。由于对指针的取值，变量score（而不是pScore）也被修改为85.0。


25
用fgets()从数据文件中读取字符串
216
整数占用的内存空间通常大于1字节
214
不能修改常量
213
不用strcpy()给字符串赋值
214
BOTTOMLESS PIT
不要用内置函数来填充字符型指针的位置
217



第 25 章

数组和指针有什么不同

它们在C语言中是相同的

本章内容

数组名是指针
212

在数组中取值
213

字符和指针
214

小心字符串的长度
214

指针数组
216

趣闻轶事
日常生活中的例子，比如日历就是存储了12个元素的数组。

本章将讲解C语言的数组和指针变量怎样共享许多法则。事实上，数组是特殊类型的指针。由于它们的相似性，你可以用指针符号来获取数组的值，也可以用数组符号来访问某指针指向的值。

也许学习数组和指针是怎样相似的最重要的原因是字符串的处理。通过组合指针符号（用取值运算符）和数组符号（用下标），你可以存储字符串列表并像引用其他数据类型的数组值一样引用它们。

另外，一旦掌握了堆（heap）——内存中的一个特殊位置，在下一章将介绍——你将看到指针是获取存放数据的堆内存的唯一方法。

25.1　数组名是指针

数组名只不过是指向第一个数组元素的指针。尽管如此，数组名并不完全是一个指针变量（variable）。数组名是指针常量（pointer constant）。下面的语句定义了一个整型数组并对其初始化：

```
int vals[5] = {10, 20, 30, 40, 50};
```

你可以通过下标符号引用数组。这是早就已经知道的。然而，C语言不仅仅是把下标与内存中的值进行关联。C语言设置一个指向数组的指针并命名为vals。你永远不能修改vals的内容，它就像一个地址被C语言锁定的固定的指针变量。图25-1表明了当你定义并初始化vals时，C语言真正做的事。

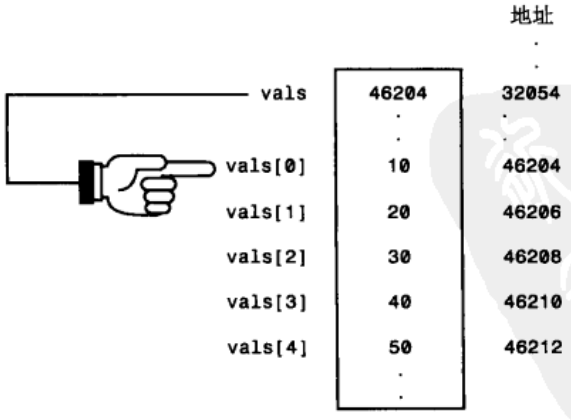


图25-1　数组名是指向数组的第1个值的指针

因为数组名是一个指针（不能被修改），所以你可以像下面这样打印数组的第1

个值：

```
printf("The first value is %d.\n", vals[0]);
```

但是本章要强调的是，你也可以像下面这样打印数组的第1个值：

```
printf("The first value is %d.\n", *vals);
```

你很快就会看到，下面也是访问vals[0]的等价写法：

```
printf("The first value is %d.\n", *(vals+0));
```



警告 数组是一个固定的常量指针这一事实是你不能把数组名放在赋值运算符左边的原因。你不能修改常量。记住，只有当你首次定义数组时，C语言才允许你这么做，因为此数组还没有固定在一个指定的地址上。

25.2 在数组中取值

因为数组名只不过是指向数组第1个元素的指针，所以如果你想获取第2个值，你只需要给数组名加上1并对其取值。下面的printf()语句：

```
printf("The first array value is %d.\n", vals[0]);
printf("The second array value is %d.\n", vals[1]);
printf("The third array value is %d.\n", vals[2]);
printf("The fourth array value is %d.\n", vals[3]);
printf("The fifth array value is %d.\n", vals[4]);
```

与下面这些做了完全相同的事：

```
printf("The first array value is %d.\n", *(vals + 0));
printf("The second array value is %d.\n", *(vals + 1));
printf("The third array value is %d.\n", *(vals + 2));
printf("The fourth array value is %d.\n", *(vals + 3));
printf("The fifth array value is %d.\n", *(vals + 4));
```

如果vals是一个指针常量（而它的确是），而且指针常量存放了一个数字，它是数组第1个元素的地址，在对vals取值之前给vals加上1或2（或任何值）就是给val指向的内存地址加上1或2。



线索 如果你想知道这些的重要性。很快你将会看到C语言的指针符号如何使它看起来像有字符串变量一样。



请跳过这里，这有点儿难　你可能还记得，整型通常占用多于1字节的存储空间。前面的printf()语句给vals中的地址加上1以得到下一个取值的内存位置，但是C语言在这里帮你做了些工作。当你给int型指针加1时，C语言加上1个int大小（当给double型指针加1时，加上一个double大小）。表达式*(vals + 2)告诉C程序你想要vals指向列表中的第3个整数。

25.3　字符和指针

下面的两语句在内存中设置了几乎相同的内容。唯一的不同是在第2条语句中，pName是一个指针变量（variable），而不是一个指针常量：

```
char name[] = "Andrew B. Mayfair";   /* name points to A */
char * pName = "Andrew B. Mayfair";  /* pName points to A */
```

因为pName是一个指针变量，你可以把它放在赋值运算符的左边！因此，如果你想给字符指针赋一个新字符串值，不必总是用strcpy()。字符指针只指向字符串中的第1个字符。然而，%s和所有的字符串函数处理字符指针像处理字符数组一样容易（二者是同样的东西），因为这些函数知道在null零处停下。

要给name数组一个不同的值，不得不使用strcpy()或者一次一个字符地赋值字符串，但是要使pName指向一个不同的值，只要这样做：

```
pName = "Theodore M. Brooks";
```



线索　字符串赋值能起作用的唯一原因是C语言把程序中所有的字符串字面量放到内存中的某个地方，然后在程序中用地址替换它们。C语言没有真正地把Theodore M. Brooks放到pName中，因为只能存放地址。C语言只是把Theodore M. Brooks的地址存放到pName中。



说明　你现在已经知道了一种不用strcpy()进行字符串赋值的方法了。

25.4　小心字符串的长度

可以在刚才描述的字符数组中存储字符串字面量。赋予的新字符串可以比原来

的字符串更长或更短。这很好，因为你可能还记得你不能把一个比初始预留的数组更长的字符串存储到字符数组中。

然而，你必须十分小心，不要让程序存储比你用字符指针指向的第1个字符串更长的字符串。这有一点复杂，但是本章尽量简洁明了地讲解。不要像下面这样设置字符指针：

```
main()
{
char * name = "Tom Roberts";
/* Rest of program follows */
```

然后像这样用gets()让用户输入一个新的字符串：

```
gets(name);  /* Not very safe */
```

这条语句的问题是用户可能输入一个比Tom Roberts更长的字符串。尽管字符指针可以指向任意长度的字符串，但是gets()函数和scanf()、strcpy()和strcat()不知道传入的是一个字符指针。因为它们可能被传入不改变位置的字符数组，所以这些函数把新创建的字符串直接映射到name中字符串的位置上。如果输入的字符串比name中的更长，可能会改写其他的数据区域。



警告 是的，这可能有一点不好理解。在你对指针和数组更了解之后，你可能得再次阅读这一节。

如果你想把字符串字面量赋给指针并且仍然保有使用数组的安全性，这样就可以用字符指针来获取用户的输入——你可以用一点技巧来实现这一点。

如果想把用户的输入存储到一个指针指向的字符串中，那么首先要为输入的字符串预留足够的存储空间。最简单的办法是预留一个字符数组并把字符指针赋为数组的第1个元素。

```
char input[81];  /* Holds a string as long
                    as 80 characters */
char *iptr = input;  /* Also could have done this:
                        char *iptr = &input[0]; */
```

现在你可以用指针来输入字符串了：

```
gets(iptr);  /* Makes sure that iptr points to
                the string typed by the user */
```

只要用户输入的字符串不超过81个字节。有一个很好的字符串输入函数可以使

用，它可以确保输入的字符串不会超过81个字符（包含null零）。如果想限制从用户处接收到的字符数，就使用fgets()。fgets()像gets()一样工作，唯一不同的是要指定一个长度参数。下面的语句说明了fgets()的使用方法：

```
fgets(iptr, 81, stdin);  /*Gets up to 80 chars and adds null
                           zero */
```

第2个值是你想从用户的输入中保存的最大字符数。记住，你始终要为字符串的null零留1个数。指针iptr能指向不超过81个字符的字符串。如果用户输入的字符串少于81个字符，iptr可以正常地指向这个字符串。然而，如果用户输入200字符的字符串，iptr只能指向前80个字符以及在第81个位置上的null零。其余的输入都被忽略了。



线索　你可以用fgets()从数据文件中读取字符串。fgets()的第3个值可以是磁盘文件指针，在本书的后面你将学到磁盘指针。现在，用stdin作为传给fgets()的第3个值，这样fgets()从键盘而不是从其他地方获取输入。

你也可以使用像下面这样的赋值语句把字符串字符量赋给指针：

```
iptr = "William Harper Littlejohn";
```

25.5　指针数组

如果你想使用一组指针，就创建一个指针数组。定义一个指针数组与定义任何其他类型的数据数组一样简单，只是你必须在数据类型名的后面加上*运算符。下面的语句定义了一个包含25个整型指针的数组和一个包含25个字符型指针的数组。

```
int * ipara[25];   /* 25 pointers to integers */
char * cpara[25];  /* 25 pointers to characters */
```

字符数组是最有趣的，因为你可以在数组中存储字符串的列表。更准确地说，你可以指向不同的字符串。下面的程序说明了两件事：如何在定义时初始化字符串数组以及如何使用for循环打印它们：

```
#include <stdio.h>
main()
{
  int i;
  char * names[5] = {"Joe Swadley", "Richard Wikert",
                     "Keith Miller", "Dean Davenport",
                     "Stacy Wiquet"};
```

```
  for (i = 0; i < 5; i++)
    { printf("Name %d: %s\n", i, names[i]); }
  return 0;
}
```

下面是程序的输出：

```
Name 0: Joe Swadley
Name 1: Richard Wikert
Name 2: Keith Miller
Name 3: Dean Davenport
Name 4: Stacy Wiquet
```

图25-2说明了程序是如何在内存中设置names数组的。每一个元素只不过是一个字符型指针，包含了不同人名的地址。理解names不存放字符串这一点很重要，它只存放指向字符串的指针。

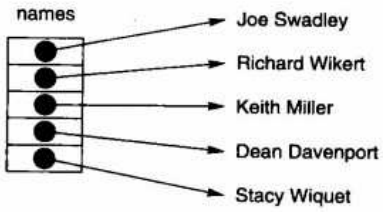


图25-2 names数组包含指向字符串的指针

看，即使C语言中没有像字符串数组这样的东西（因为没有字符串变量），但是把字符型指针存储在names中就可以使程序表现得像names是一个字符串数组一样。



奖励

- 如果你想直接赋值字符串字面量，就使用字符型指针。
- 使用数组下标符号或者指针取值运算符来访问数组和指针的值。
- 休息一下！还有很多内容。



陷阱

- 不要用内置函数来填充字符型指针的位置，除非字符型指针初始设置为指向一个很长的字符串。

25.6 小结

本章讲解了数组和指针的相似性。数组名其实只是指向第1个元素的指针。

与指针变量不同，数组名不能修改。这是数组名不能出现在赋值运算符左边的主要原因。

使用指针比数组有更多的灵活性。你可以直接把一个字符串字面量赋给一个字符型指针变量，而你必须用strcpy()函数来把字符串赋给数组。在以后的C编程中，你还将看到指针变量的很多用法。

1. 代码范例

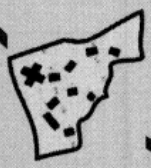
```
char * days[7];
days[0] = "Sunday";
days[1] = "Monday";
days[2] = "Tuesday";
days[3] = "Wednesday";
days[4] = "Thursday";
days[5] = "Friday";
days[6] = "Saturday";
printf("Enter a number from 1 to 7: ");
scanf(" %d", &dayNum);
if (dayNum >= 1 && dayNum <= 7)
  { printf("That day is %s\n", days[dayNum - 1]); }
else
  { printf("You didn't enter a good number.\n"); }
```

2. 代码分析

这段代码创建了包含7个指针变量的数组并赋予它们从周一到周日的字符串字面量。可以在定义days的时候赋予其所有的字符串，但这段代码每次赋予其一个字符串，是为了说明如何给数组赋值。

程序然后要求输入一个数字。如果用户输入一个从1到7的数，那么匹配的元素就会被输出。因为数组元素从0开始，所以用户输入的数字必须减去1才能匹配。

26



不要把变量名分配给堆内存
222



每个分配的内存
都是连续的
226



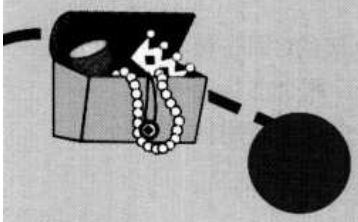
需要大量练习才能掌握堆
224



使用malloc()函数
227



不要总是用普通数组
229



第 26 章

有效管理内存空间

堆管理

本章内容

使用堆
222

为什么需要堆
223

分配堆
224

堆内存分配失败
227

释放堆内存
227

多次分配
228

并非只有C语言初学者会对本章的主题感到困惑。即使是高级的C程序员在处理堆（heap）的时候也会犯难。堆是电脑中未使用的内存的集合。剩余的内存——在你的程序、程序的变量和操作系统的工作空间之后——组成了电脑上可用的堆空间，如图26-1所示。



图26-1　堆是未使用的内存

在很多时候你要访问堆，因为有时候你的程序需要比你初始在变量和数组中定义的更多的内存。本章教你为什么以及怎样使用堆内存，而不用变量。



线索　你不能把变量名分配给堆内存。访问存储在堆内存中数据的唯一方法是通过指针变量。难道你不高兴你已经学过指针变量了？不知道指针，你就不能学习堆。



说明　自由的堆内存称为自由的堆（free heap）或未分配的堆（unallocated heap）内存。任何一次被程序用过的堆称为已分配的堆（allocated heap）。程序在执行时可能使用不同数量的堆空间。到目前为止，本书还没有一个程序用到堆。

26.1　使用堆

既然你已经学习了什么是堆——连续内存中未使用的部分——暂时不去想已学过的内容！如果你只是把堆看成是一大堆自由的内存，那么你将能更快地领会如何使用堆。下一段将解释原因。

你将在程序运行时分配（使用）和解分配（释放）堆内存。当你请求堆内存时，你不能准确地知道新的内存从堆的什么地方来。因此，如果程序中的一条语句获取了堆内存，然后正好下一条语句也获取另一部分堆内存，那么第二部分内存可能不会在物理上紧跟在第一部分内存的后面。



线索 正如用铲子挖一大堆土一样，一铲子土不可能正好在前一铲子土的下面。另外，当你把一铲子土扔回堆中时，那些土不会正好回到原来的地方。尽管这个比喻可能看起来有点歪曲了计算机内存的概念，但是如果你像看待泥土堆一样地看待堆内存，你会发现你将更好地理解堆：你不能准确地知道分配和解分配的内存来自何处或者回到何处。你只知道这些内存从堆中来而且回到堆中去。

如果你一次分配了10个字节的堆内存，这10个字节将会是连续的。需要知道的重点是你分配的下一部分堆内存不一定紧跟前一个的后面，因此你不应该指望这样的事会发生。

操作系统和你的程序一起使用堆内存。如果你在联网的计算机上工作，或者使用的是多任务操作环境如Windows，那么其他任务可能和你的程序一起使用堆内存。因此，在你的两条堆分配语句之间，可能有另一个程序进行了分配和解分配的操作。

你不得不跟踪你所分配的内存。可以用指针变量来实现这一点。例如，如果想在堆上分配20个整数，你可以使用一个整型指针。如果想在堆上分配400个浮点值，可以使用浮点型指针。这些指针将始终指向所分配的内存的第1个值。因此，一个指针指向所分配的堆的起始处。如果想访问在堆上第1个值后面的内存，你可以使用指针符号或数组符号到达所分配的堆的其他部分。看，上一章中的指针/数组讨论派得上用场了！

26.2 为什么需要堆

好的，在详细学习怎样分配和解分配堆内存之前，你可能更想知道为什么要使用堆。毕竟，到目前为止你已经学过的变量、指针和数组已经足够满足程序数据的需要了。

堆内存不会总是替代你已经学过的变量和数组。关于变量的问题是你必须预先准确地知道需要什么类型的变量及其个数。请记住，你必须在使用之前定义所有的变量。如果你定义了一个数组来存放100个顾客ID，但是用户有101个顾客要输入，那么你的程序就不能在运行时扩展数组的元素个数。一些程序员（像你）不得不去修改数组的定义并重新编译程序以使数组能够存放更多的值。

然而，使用堆内存，你就不必预先知道需要多少内存。像手风琴[①]一样，程序使用的堆内存可以按照需要增加或收缩。如果你需要另外100个元素来存放一批新的顾客，程序可以在运行时分配新的内存，而不用重新编译。



警告　本章不可能讲解所有有关堆的知识。掌握堆还需要大量的练习，而且在现实中，真正需要堆的程序超过了本书的范畴。但是，当你学完本章，你将会扎实地理解如何访问堆，而且由于所采用的教学方法，你将比从大多数书上学到的理解得更扎实。我可能会带你进行一个危险的旅行，但总能把你安全带回。

趣闻轶事
文字处理软件几乎没有不使用堆的。

商业软件（如电子数据表软件和文字处理软件）必须大大地依赖于堆。毕竟，设计软件的程序员无法准确知道电子表格或文字处理文档有多少或多小。因此，当你向电子表格或文字处理器中输入数据时，潜在的程序要分配更多的数据。程序在你输入时，可能不是一次只分配1个字节，因为当一次只分配1个字节时，内存分配可能不是很高效。更可能的是，程序将分配一大块内存，如100字节或500字节。

为什么程序员不能只是分配极大的能够存放极大的电子表格或文档的数组，而要使用堆呢？首先，内存是计算机中最珍贵的资源之一。当处在联网的且带视窗的环境下时，内存变得更加珍贵。程序不能为偶然出现的需要大量内存的情况分配极大的数组。所有的内存只能由你的程序独自使用，其他程序不能访问已分配的内存。



说明　堆使你的程序只使用所需的内存。当你的用户需要更多的内存时（例如，输入更多的数据），程序可以分配内存。当用户使用完那些内存时（如在文字处理器中清除一个文档并创建一个新文档），你可以释放分配的内存，使其可供其他需要内存的程序使用。

26.3　分配堆

你只需学习两个新函数就能使用堆了。`malloc()`函数（用来分配内存）分配堆的内存，而`free()`函数释放分配堆的内存。

① 手风琴是一种既能够独奏，又能伴奏的簧片乐器。手指与风箱的巧妙结合，能够演奏出多种不同风格的乐曲。——编者注



线索 在所有使用malloc()和free()函数的程序中都要包含STDLIB.H头文件。

单讲malloc()函数可能不是很好理解。也许看一个malloc()函数的例子会让你更好理解。假设你在为本地的天气预报编写一个计算平均温度的程序。用户输入的温度读数越多，预报的结果就会越精确。你决定分配10个整数来存放前10个温度读数。如果用户想输入更多读数，你的程序还能够分配另外10个整数，依此类推。

你首先需要一个指向10个堆中的值的指针。因为这些值是整数，所以你需要整型指针。你得像下面这样定义整型指针：

```
int * temps;  /* Will point to the first heap value */
```

下面是如何用malloc()在堆上分配10个整数：

```
temps = (int *) malloc(10 * sizeof(int));  /* Yikes! */
```

这么多代码只是为了得到10个整数。当你把这行代码分成几段来看时，它实际上非常容易理解。malloc()函数只需要一个值——想要分配的字节数。因此，如果你想要10个字节，可以这样写：

```
malloc(10).
```

问题是前面的描述需要的不是10个字节，而是10个整数。10个整数需要多少字节的内存？10个还是20个？答案当然是不确定的。只有sizeof()确切地知道。

因此，如果你想分配10个整数，就必须告诉malloc()函数你想要分配10组字节，每一组都足够存放一个整数。因此，前面的代码行包含了如下的malloc()函数调用：

```
malloc(10 * sizeof(int))
```

这部分代码让malloc()函数在堆上分配或者留出10个连续的整数位置。从某种程度上说，计算机在这10个整数位置周围加上围墙，这样后面的malloc()调用就不能侵入这块已分配的内存。既然你已经掌握了malloc()语句的后半部分，其他的就不成问题了。malloc()的前半部分十分简单。

malloc()函数总是执行下面两个步骤(假设有足够的堆内存来满足分配请求)：

(1) 分配所请求的字节数并确保其他程序都不能改写这块内存直到程序将其释放；

(2) 把指针指向第1个分配的值。

图26-2说明了前面的温度malloc()函数调用的结果。如你从表中所见，堆中现在包含了一块由10个整数组成的受保护的区域，而且名为temps的整型指针变量指向第1个整数。后面的malloc()函数调用将访问堆的其他部分并且不会与已分配的10个整数相交。

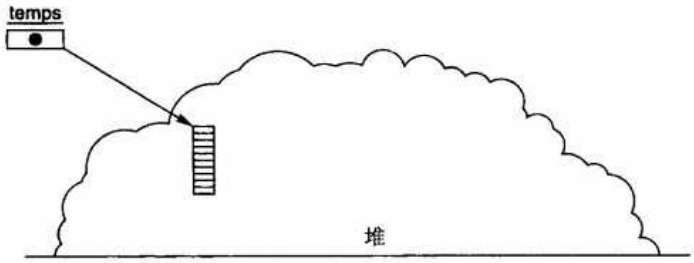


图26-2　在分配了10个整数之后



请跳过这里，这有点儿难　可以对刚分配的10个整数做些什么？把它们当作数组看待就好！可以通过使用temp[0]、temp[1]等存储数据。从上一章中知道，用数组符号可以访问连续的内存，即使内存以指针开始。另外，记住每一组已分配的内存都是连续的，所以10个整数将会是一个接一个的，就像一个包含10个元素的整型数组一样。

仍然有一个关于malloc()分配的小问题。温度malloc()的左边部分还没有解释。(int *)有什么作用？

(int *)是一个强制类型转换。你已经在本书中见过其他类型的强制类型转换。为了把一个float型的值转换成int型，你得在浮点值前面加上(int)，像这样：

```
aVal = (int)salary;
```

强制类型转换中的*意味着此强制类型转换是一个指针强制类型转换。malloc()函数总是返回一个字符型指针。如果你想用malloc()来分配整数、浮点数或除了char之外的任何数据类型，就不得不对malloc()进行强制类型转换，这样接收分配的指针变量（如temps）就能接收到正确的指针数据类型。temps是一个整型指针，你不应该把temps指向malloc()分配的内存，除非你将malloc()转换为整型指针。因此，前面malloc()的左边部分只是告诉malloc()指向分配后的第1个值的是一个整型指针，而不是默认的字符型指针。



说明 除了在main()函数的顶部定义一个数组，通过使用malloc()函数还能有什么好处？首先，可以在程序的任何地方使用malloc()函数，而不仅仅在定义变量和数组的地方。因此，当你的程序为100个double值做好了准备，你就可以分配100个double值了。如果使用普通数组，你需要这样的语句：

```
double myVals[100];  /* A regular array of 100 doubles */
```

出现在main()函数的顶部。这100个double值将在程序的生命周期中始终占用宝贵的内存资源，即使程序只在很短的时间内需要它们。有了malloc()函数，你只需要定义指向所分配的内存顶部的指针，而不是整个数组。

26.4 堆内存分配失败

在极端情况下，可能没有足够的堆内存来满足malloc()函数的请求。原因有可能是用户的电脑没有很多内存，也可能是另一个程序已经使用了很多的内存，或者你的程序已经在前面分配了太多内存。如果malloc()分配内存失败了，它的指针变量将指向null值（0）。因此，许多程序员在malloc()语句后面加上一条if语句，像下面这样：

```
temps = (int *) malloc(10 * sizeof(int));  /* Yikes! */
if (temps == 0)
  { printf("Oops! Not enough memory.\n");
    exit(1);    /* Exit the program early */
  }
/* Rest of program continues */
```

如果malloc()分配内存成功了，temps会包含一个有效的地址，它指向所分配的堆的起始处。如果malloc()分配内存失败了，temps就指向无效的地址0（堆内存从来不从地址0开始）并且在屏幕上打印出错误消息。



线索 通常，程序员像下面这样使用非（not）运算符!，而不是使用与0作比较的方式。因此，前面的if测试也可以写成这样：

```
if (!temps)      /* Means, if not true */
```

26.5 释放堆内存

用完堆内存后，还要把它还给系统。使用free()来实现。free()要比malloc()

简单得多。要释放前面分配的10个整数，可以像下面这样使用free()函数：

```
free(temps);    /* Gives the memory back to the heap */
```

如果最初分配了10个值，就会释放10个值。如果为temps分配内存的malloc()分配了1000个值，那么将会释放所有的1000个值。一旦释放，就不能再把这些内存要回来了。记住，free()把已分配的内存扔回堆中，并且一旦扔回去了，这些内存就可能被另一个程序夺取（始终记住泥土堆的比喻）。如果在前面free()后面使用temps，你就有改写内存的风险，甚至可能锁死你的电脑，不得不重新启动。

如果释放已分配的内存失败，那么一旦程序结束，操作系统将会回收这些内存。然而，忘记调用free()首先违背了使用堆内存的目的。堆的目的是使你的程序能够在需要时分配内存并且在用完之后释放分配的内存。

26.6　多次分配

通常，指针数组帮助你分配许多不同组的堆内存。回到天气预报员的问题，假设预报员需要输入几个城市的历史温度读数。对每个不同的城市，预报员都有不同数量的读数。

指针用来处理这种问题。下面的语句说明了如何分配包含50个指针的数组：

```
int * temps[50];    /* 50 integer pointers */
```

这个数组不是存放50个整数（由于定义中的取值运算符），而是存放50个指针。第1个指针叫做temp[0]，第2个指针是temp[1]，依此类推。每个数组元素（每个指针）可以指向不同的已分配的堆内存集。因此，尽管这50个指针数组元素必须为main()函数定义，但是当需要额外的内存时，也可以分配和释放所指向的数据。

看看下面的代码段，它可能被预报员使用：

```
for (ctr=0; ctr<50; ctr++)
  { puts("How many readings for the city?");
    scanf(" %d", &num);

    /* Allocate that many heap values */
    temps[ctr] = (int *)malloc(num * sizeof(int));

    /* It is here that the code would ask for each
       temperature reading for this city */
  }
/* Calculations would go here */
```

```
/* Once done with the heap memory, deallocate it */
for (ctr=0; ctr<50; ctr++)
  { free(temps[ctr]); }
```

当然，这样的代码需要大量的数据输入。那么值最可能是来自一个历史的磁盘文件而不是来自用户输入。但是，这段代码使你看到了通过使用堆从而可用的高级数据结构。另外，实际的程序通常不会像本书中经常看见的只有20行左右的代码。尽管实际的程序不一定比这里的难，但是它们通常有许多页长。在程序中，一些部分可能需要额外的内存，而另一些部分不需要那些内存。堆使你可以高效地使用内存。

图26-3说明了在分配temps数组内存的过程中（在50个malloc()调用中的前4个调用之后）堆内存的样子。如你所见，temps属于程序的数据区域，但由每个temps元素指向的内存却属于堆。当你不再需要额外的内存时，就可以释放由temps指向的数据。

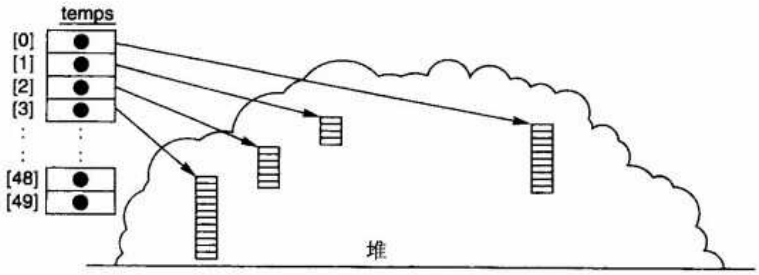


图26-3 每个temps元素指向堆的不同部分



奖励

- 使用malloc()和free()函数来分配和释放堆内存。
- 通过在malloc()的括号中使用sizeof()运算符来精确地告诉malloc()每次必须分配多大的内存。
- 在函数的顶部只分配指针变量和其他变量。当你需要除了简单的循环计数器和总数之外的数据值时，可以把数据放在堆中。
- 如果必须跟踪几块堆内存，就使用指针数组。每个数组元素都可以指向不同数量的堆空间。
- 检查以确保malloc()正确地工作。如果分配内存失败，malloc()就返回0。



陷阱

- 不要总是依赖普通数组来存放程序的数据。有时候，程序只是在短时间才需

要的数据，这种情况下使用堆能够更好地利用内存资源。

- 如果此刻你还没有掌握堆，不要气馁。堆的使用被认为是C语言编程中最困难的部分，不过你已经在本章中看到，堆的使用并没有那么难。

26.7　小结

你是否完全混乱了，或者你还想知道关于malloc()和free()函数的更多信息。内存分配不像puts()函数调用那样简单，但是你已经看到所有都与malloc()和free()有关。作为一名C程序员，你还有很长的路要走，而在精通malloc()之前，它很可能会不止一次地使你感到困惑。

但是，分配内存和释放分配的内存比在你刚开始学习这一章时对你更有意义了。我知道，你在学这一章之前从来没有听说过分配内存！当你在C语言编程过程中取得进步时，还需要学习其他程序员是如何使用malloc()和free()的。

作为回顾，malloc()为程序分配堆内存。可以通过指针变量访问堆，还可以通过使用数组符号基于malloc()分配的指针来到达已分配的内存的其他位置。

用完堆内存后，还要用free()函数来释放分配的内存。free()把内存扔回堆中，这样其他程序就可以使用它了。

1. 代码范例

```
#include <stdlib.h>
int * iPtr;
float * fPtr;
double * dPtr;
iPtr = (int *)malloc(100 * sizeof(int));
if (!iPtr)
  { puts("The integer allocation failed.");
    exit(1);
  }
fPtr = (float *)malloc(50 * sizeof(float));
if (!fPtr)
  { puts("The float allocation failed.");
    exit(1);
  }
dPtr = (double *)malloc(450 * sizeof(double));
if (!dPtr)
  { puts("The double allocation failed.");
    exit(1);
  }
/* You now have allocated a total of 600 heap values */
for (ctr=0; ctr<100; ctr++)
```

```
  { iPtr[ctr] = ctr; }  /* Store 0 through 99 */
for (ctr=0; ctr<50; ctr++)
  { *(fPtr+ctr) = (float)ctr; }  /* Store 0 through 49 */
for (ctr=0; ctr<450; ctr++)
  { iPtr[ctr] = (double)ctr; }  /* Store 0 through 449 */
free(iPtr);
free(fPtr);
free(dPtr);
```

2. 代码分析

这段代码分配了3组堆内存。iPtr整型指针变量指向100个分配的整数值中的第1个值，fPtr指向50个已分配的浮点值中的第1个值，而dPtr指向450个已经分配的双精度浮点值中的第1个值。

在每个malloc()后面，都检查了指针的值以确保分配了所需的内存。

然后，3个for循环把从0开始的序号存储到已分配的内存中。整数循环计数器ctr用来初始化这3段堆，于是在把ctr的值存储到堆之前还要用float和double强制类型转换把它转换为正确的数据类型。使用指针符号而不是数组符号来初始化浮点值的堆。

用完堆内存之后，记得要释放它。3个free()函数调用释放由前面3个malloc()调用所分配的全部600个值。

27



结构体就像卡片盒
235



把结构体看作是单独数据类型的组合
234



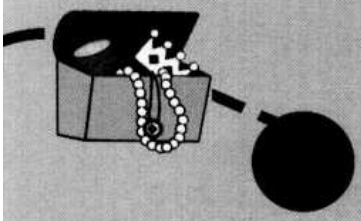
定义结构体的变量
235



把这些数据分组称为结构体
234



不要把成员名当作变量使用
240



第27章

结 构 体

使用结构体

本章内容

定义结构体
234

在结构体变量中存放数据
238

数组和指针非常适用于值的列表，但是这些值必须具有相同的数据类型。不过，有时候还需要把不同的数据类型放在一起并作为一个整体来对待。

顾客记录就是一个很好的例子。对于每个顾客，都得记录下名字（字符数组）、余额（双精度浮点值）、地址（字符数组）、城市（字符数组）、州（字符数组）和邮政编码（字符数组或长整数）。尽管你希望能够初始化并打印顾客记录中的一个字段，但是你也希望把顾客记录当作一个整体来访问，比如当你把它写到磁盘文件（在下一章中介绍）上时。

C语言的结构体（structure）可以用来对出现在顾客记录中的这样的数据进行组织，并且能够访问所有的单个部分（称为成员，member）。如果这样的数据会出现多次并且有许多顾客，就需要用结构体数组了。



说明　其他编程语言有等价的数据分组，称为记录（record）。然而，C语言的设计者想把这些数据分组称为结构体（structure）。

趣闻轶事
QBasic在有维度的类型中存储结构体。

很多时候，C语言的结构体中存放的数据可能是你存放在卡片盒中3×5大小的卡片上的数据。没有计算机，公司可能会维护一个卡片盒，里面的卡片上包含了顾客的名字、余额、地址、城市、州和邮政编码，就像前面描述的顾客结构体那样。在本章后面的内容中，你将看到C语言的结构体是如何存储在内存中，以及更多与卡片的相似点。

27.1　定义结构体

你所要做的第一件事是准确告诉C语言你的结构体的样子。当定义内置数据类型的变量（如int）时，你不需要告诉C编译器int是什么，因为它已经知道了。但是，当你想定义一个结构体时，你必须首先准确地告诉C语言你的结构体的样子。然后，而且只有在这之后，你才能为结构体定义变量。



请跳过这里，这有点儿难　试着把结构体看成是一些单独数据类型的组合。整个结构体有一个名字并且可以从总体上看成是一个值。结构体的一个成员是内置的数据类型，比如int和char型数组，可以表示年龄和名字。如果愿意，你可以访问一个成员。



线索 结构体不仅像卡片盒，而且你会发现它很像尚未填空的表单。一张空表（比如你申请信用卡可能要填写的）本身没有用。如果信用卡公司打印了10000份表单，并不意味着他们有了10000个客户。只有当某人填写了一张表单后，才有了一个客户，而且只有当用你描述的结构体来定义变量时，C语言才会为结构体变量分配内存。

趣闻轶事
在其他编程语言中把结构体的成员叫做域。

为了定义int型变量，只需要这样写：

```
int i;
```

不需要首先告诉C编译器int是什么。为了定义结构体变量，你必须首先定义结构体的样子并给它一个数据类型名，如customer。在定义了结构体的格式之后，可以定义变量。

struct语句定义了结构体的外观（或布局）。下面是struct的语法格式：

```
struct [structure tag] {
  member definition;
  member definition;
   :
  member definition;
};
```

再强调一下，struct只定义了结构体的布局或外观。*structure tag*是你给这个特定结构体的外观起的名字，但是它与随后要创建的结构体变量名没有任何关系。在定义了结构体的格式之后，就可以定义变量了。

*member definitions*只不过是普通的内置数据类型定义，如int age;。但是，你是在定义成员而不是变量，达到了给结构体的特定部分起一个名字的效果。



警告 你可以在struct声明语句的同时定义变量，但是大多数程序员不这么做。如果你想在声明结构体格式的同时为它定义变量，可以在struct语句结束的分号前面插入一个或多个变量名。



说明 结构体还有很多内容要学习。下面的例子有助于你理解。

假设你正在编写一个程序来跟踪零售电脑的存货。你需要记录电脑的厂商、模型、磁盘空间大小（MB）、内存空间大小（MB）、数量、成本和零售价。

首先你必须用struct来定义一个结构体。下面是一种好的写法：

```
struct invStruct {
  char manuf[25];   /* Manufacturer name */
  char model[15];   /* Model code */
  int  diskSpace;   /* Disk space in megabytes */
  int  memSpace;    /* Memory space in megabytes */
  int  quantity;    /* Number in the inventory */
  float cost;       /* Cost of computer */
  float price;      /* Retail price of computer */
};
```

图27-1表明了这个结构体格式的样子。

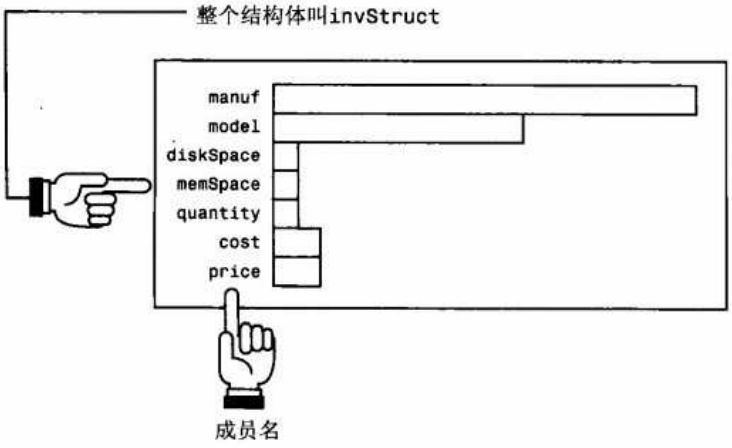


图27-1　invStruct结构体的格式

前面的结构体定义并没有定义7个变量！它只是定义了一个结构体数据类型。请记住，在定义整型变量之前不需要告诉C编译器整数的样子，但是，在定义结构体数据类型的变量之前，必须告诉C编译器invStruct的样子。前面的struct语句告诉C编译器invStruct是怎样的。一旦C编译器知道了结构体的格式，C语言就可以在用户准备好定义变量时定义这种结构体格式的变量。

如果你创建的结构体可能还要被再次使用，可以考虑把它放在自己的头文件中，或者与其他公用的结构体一起放在一个头文件中。在需要用它的源文件中用#include把头文件包含进来。如果一旦需要修改结构体的定义，只需要在一个地方——在它的头文件中——查找并修改它。



线索　上面的建议会带给你很大帮助！

通常，程序员把结构体的声明（比如前面invStruct的声明）放在main()的前面，然后在main()和main()下面的其他函数中定义这个结构体的变量。为了结构体的变量，你必须做与你创建任何其他类型的变量相同的事情：把结构体的名字写在变量列表的前面。因为不存在名为invStruct的数据类型，你必须告诉C编译器invStruct是一个struct名。可以像这样定义3个结构变量：

```
#include "c:\inv.h"  /* Includes the structure definition */
main()
{
  struct invStruct item1, item2, item3;
  /* Rest of program would follow */
```

现在就有了3个可以存放数据的变量。这3个变量是名为item1、item2和item3的结构体变量。如果你想定义500个结构体变量，可以用数组：

```
#include "c:\inv.h"  /* Includes the structure definition */
main()
{
  struct invStruct items[500];
  /* Rest of program would follow */
```

请记住，如果你用这种方法，那么结构体定义必须在INV.H头文件中。否则，你必须把结构体定义直接写在定义结构体变量之前，像这样：

```
struct invStruct {
  char manuf[25];  /* Manufacturer name */
  char model[15];  /* Model code */
  int  diskSpace;  /* Disk space in megabytes */
  int  memSpace;   /* Memory space in megabytes */
  int  quantity;   /* Number in the inventory */
  float cost;      /* Cost of computer */
  float price;     /* Retail price of computer */
};
main()
{
  struct invStruct items[500];
  /* Rest of program would follow */
```

只要struct定义出面在main()之前，你就可以在程序其他部分的任何函数中定义invStruct结构体变量。本书的最后一部分讲解如何编写不只包含main()函数的程序。

也许你需要3个结构体指针，而不是结构体变量？可以像下面这样定义它们：

```
main()
{
  struct invStruct *item1, *item2, *item3;
  /* Rest of program would follow */
```

item1、item2和item3现在指向3个结构体变量。然后你可以为结构体预定堆内存而不使用实际的变量。下面的3条语句预定了3个堆结构区域，并把item1、item2和item3指向这个3个堆的值：

```
item1 = (struct invStruct *)malloc(sizeof(invStruct));
item2 = (struct invStruct *)malloc(sizeof(invStruct));
item3 = (struct invStruct *)malloc(sizeof(invStruct));
```

27.2　在结构体变量中存放数据

一个新的运算符——*点运算符*（dot operator），使你可以给结构体变量的单个成员赋值。下面是点运算符的语法格式：

structureVariableName.memberName

点运算符的左边总是结构体变量的名字，如item1或employee[16]。点运算符的右边总是该结构体的成员名，如quantity、cost或name。点运算符只在已命名的结构变量中存放数据。如果你想在被一个结构指针变量指向的堆结构中存放数据，就必须使用*结构指针运算符*->。

下面的程序用前面展示的invStruct结构体标签定义了包含3个结构体变量的数组。假设结构体存储在INV.H头文件中，这样例子比较简短。用户被要求填充结构体变量，然后程序打印出它们。在下面两章中，你将看到如何把结构体变量输出到磁盘文件中以便长期保存。

```
/* First, include the file with the structure declaration */
#include "c:\inv.h"
#include <stdio.h>
main()
{ int ctr;
  struct invStruct items[3];  /* Array of three
                                 structure variables */
  for (ctr = 0; ctr < 3; ctr++)
    {  printf("What is the manufacturer of item #%d?\n",
              (ctr + 1));
       gets(items[ctr].manuf);
       puts("What is the model? ");
       gets(items[ctr].model);
       puts("How many megabytes of disk space? ");
       scanf(" %d", &items[ctr].diskSpace);
       puts("How many megabytes of memory space? ");
       scanf(" %d", &items[ctr].memSpace);
       puts("How many are there? ");
       scanf(" %d", &items[ctr].quantity);
       puts("How much does the item cost? ");
       scanf(" %f", &items[ctr].cost);
```

```
        puts("How much does the item retail for? ");
        scanf(" %f", &items[ctr].price);
        getchar();  /* Clears input of last newline
                       pressed for next round of input */
      }  /* Now, prints the data */
    printf("\n\nHere is the inventory: \n");
    for (ctr = 0; ctr < 3; ctr++)
      {  printf("#%d: Manufacturer: %s", (ctr + 1),
              items[ctr].manuf);
        printf("\nModel: %s", items[ctr].model);
        printf("\nDisk: %d megabytes\t", items[ctr].diskSpace);
        printf("Memory: %d megabytes\t", items[ctr].memSpace);
        printf("Quantity: %d units\n", items[ctr].quantity);
        printf("Cost: $%.2f\t", items[ctr].cost);
        printf("Selling price: $%.2f\n\n", items[ctr].price);
      }
    return 0;
  }
```

如果你想在堆上存储结构体，就不能使用点运算符了，因为点运算符只能用于变量名。使用->在堆结构中存储数据。->要求左边是一个指针，而右边是一个成员名。下面的程序与前面的程序等价，只是用堆和->代替了结构体变量和点运算符。

```
/* First, include the file with the structure declaration */
#include "c:\inv.h"
#include <stdio.h>
#include <stdlib.h>
main()
{ int ctr;
  struct invStruct * items[3];  /* Array of three
                                   structure pointers */
  for (ctr = 0; ctr < 3; ctr++)
    {  items[ctr]=(struct invStruct*)malloc(sizeof(struct
       invStruct));
       printf("What is the manufacturer of item #%d?\n",
              (ctr + 1));
       /* items[ctr] is now a pointer, not a variable */
       gets(items[ctr]->manuf);
       puts("What is the model? ");
       gets(items[ctr]->model);
       puts("How many megabytes of disk space? ");
       scanf(" %d", &items[ctr]->diskSpace);
       puts("How many megabytes of memory space? ");
       scanf(" %d", &items[ctr]->memSpace);
       puts("How many are there? ");
       scanf(" %d", &items[ctr]->quantity);
       puts("How much does the item cost? ");
       scanf(" %f", &items[ctr]->cost);
       puts("How much does the item retail for? ");
       scanf(" %f", &items[ctr]->price);
       getchar();  /* Clears input of last newline
                      pressed for next round of input */
     }  /* Now, prints the data */
```

```
    printf("\n\nHere is the inventory: \n");
    for (ctr = 0; ctr < 3; ctr++)
      {  printf("#%d: Manufacturer: %s", (ctr + 1),
              items[ctr]->manuf);
         printf("\nModel: %s", items[ctr]->model);
         printf("\nDisk: %d megabytes\t", items[ctr]->diskSpace);
         printf("Memory: %d megabytes\t", items[ctr]->memSpace);
         printf("Quantity: %d units\n", items[ctr]->quantity);
         printf("Cost: $%.2f\t", items[ctr]->cost);
         printf("Selling price: $%.2f\n\n", items[ctr]->price);
      }
    return 0;
}
```



奖励

- 当你想把不同数据类型的项目组合起来时，就使用结构体。
- 在定义结构变量之前首先要声明结构体。
- 用点运算符访问结构体变量中的单个数据成员。
- 用->（结构体指针运算符）访问被指针变量指向的结构体的单个数据成员。



陷阱

- 不要把成员名当作变量使用。成员名的存在只是为了让你能够处理结构体的单个数据成员。
- 不要忘记在所有结构体定义的末尾加上分号。
- 不要混淆点运算符和结构体指针运算符。请记住结构变量必须出现在点运算符的前面，而结构体指针变量必须出现在->运算符的前面。

27.3　小结

本章讲解了结构体。结构体是一个聚合的可变的数据类型。与数组必须存放相同数据类型的值相反，结构体可以存放不同数据类型的几个值。

在使用结构体变量之前，必须用struct语句准确告诉C编译器结构体的样子。struct语句使C编译器知道结构体中有多少成员以及每个成员的数据类型。结构体变量就像多个不同数据类型的变量的组合。

1. 代码范例

```
struct telStr {
  char lastName[15];
```

```
  char initial;
  char firstname[15];
  int areaCode;
  long int phoneNum;
  char address[25];
  char city[10];
  char state[3];  /* Leaves 1 for the null zero! */
  char zip[6];
};
main()
{
  int i;
  struct telStr friends[50];
```

2. 代码分析

这段代码可能是记录你的朋友的电话号码和地址的程序的开始部分。代码首先描述了一个包含9个成员的结构体。成员由字符数组、字符型成员、整型成员和长整型成员构成。

main()开头部分的程序定义了一个包含50个结构体变量的数组。还定义了一个int型变量，是为了说明结构体变量定义的位置与其他变量定义的位置相同。50个结构体变量中的每一个都具有telStr结构体。


gary Chaval

Part 5 第五部分

用函数组织程序

本部分内容

- 第 28 章　把数据存到硬盘上
- 第 29 章　另一种保存文件的方式
- 第 30 章　用函数来组织程序
- 第 31 章　在函数间共享数据
- 第 32 章　让函数更完美



28



输出到打印机而不是屏幕
251



在fprintf()函数中
250



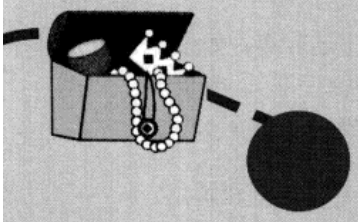
在main()前面定义文件指针
247



文件是挑剔的
246



使用文件指针
251



第 28 章

把数据存到硬盘上

顺序文件

本章内容

硬盘文件
246

打开顺序文件
247

访问顺序文件
248

目前为止，你所见过的程序都不能长期地存储数据。考虑一下这件事：如果你定义了一个整型变量，把14赋给它，然后关闭计算机（相信我并试一下），变量中就不再有14了！如果你重新打开计算机，并尝试寻找变量的值，你根本找不到了。

本章讲解如何将数据保存到硬盘上。一旦数据在硬盘上了，它将一直在那里，直到你修改或删除了它。硬盘上的数据就像磁带上的音乐一样。你可以关闭放音机，而磁带将始终存放着音乐，直到你修改了它。没有什么好的理由可以让用户多次地输入数据，比如历史的销售记录。



说明　文件对计算机数据程序来说至关重要。如果没有文件，文字处理程序还有多大用处呢？

28.1　硬盘文件

趣闻轶事
在软盘出现之前，人们使用打孔的卡片来存储数据。

硬盘把数据存放在文件（file）中。如果你已经把一个C程序保存成硬盘文件，那么你已经理解了文件的概念了。文件可以存放程序或者数据，如图28-1所示。程序在运行之前必须从硬盘装载到内存中。在你对数据进行处理之前，也必须把数据从硬盘文件装载到变量中。在数据进入硬盘文件之前，变量也存放着数据。

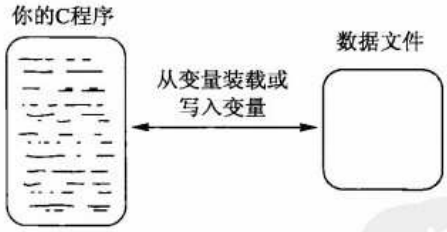


图28-1　程序文件可以处理数据文件

有两种类型的文件：顺序访问（sequential-access）和随机访问（random-access）。它们的类型决定了你可以怎样访问它们。如果处理一个顺序访问的文件，那么你不得不按照数据的顺序读或写文件。在随机访问的文件中，你可以跳跃式地读或写文件的任何位置。



线索　顺序访问的文件就像盒式磁带，而随机访问的文件就像唱片或CD。你不得不按顺序播放磁带上的歌曲（或按顺序地快进），而你可以跳过唱片或CD上的音乐。

所有的硬盘文件都有名字，这些名字遵循与程序名字相同的命名规则。无论是创建、读或修改文件中的数据，在能够使用硬盘文件之前，你必须打开（open）文件。



线索 就像档案柜一样，不能还没打开硬盘文件就使用它。与拉出一个抽屉不同，计算机把一个叫做文件指针（file pointer）的东西关联到文件上并确保合理地设置硬盘以存放指定的文件。

28.2 打开顺序文件

为了打开文件，你必须使用fopen()函数，它的描述与printf()的描述一起包含在STDIO.H中。在学习fopen()之前，你得理解文件指针的概念。



说明 文件指针的概念很容易理解。普通指针只是存放着变量中数据的地址。文件指针只是存放着正在处理的硬盘文件的硬盘地址。

你必须使用一条特殊的语句来定义文件指针。与任何变量一样，你可以把文件指针按你喜欢的方式命名。假设你想打开一个雇员文件。在fopen()前面，你必须定义一个文件指针变量。如果文件指针变量命名为fptr，下面就是文件指针的定义：

```
FILE * fptr;  /* Defines a file pointer named fptr */
```



警告 大多数C程序员在main()的前面定义文件指针。这使得文件指针成为全局的（global），意思是整个程序都可以使用这个文件。大多数其他类型的变量都是局部的（local），而非全局的。因为文件指针语句的一部分是大写的，所以FILE在某处用#define定义了。FILE定义在STDIO.H头文件中，这是当程序使用了硬盘上的数据时要包含STDIO.H头文件的主要原因。

一旦定义了文件指针，你就可以用fopen()把指针连接到某个文件。一旦指定了fopen()，你就可以在程序的其他部分使用此文件了。下面是打开名为C:\EMPS.DAT的文件的方法。如果你没有C:驱动器，就把本例中的C:改为不同的驱动器盘符。

```
#include <stdio.h>
FILE *fptr;   /* Define a file pointer */
main()
{
  fptr = fopen("C:\\EMPS.DAT", "w");
  /* Rest of program follows */
  fclose(fptr);  /* Always close the file */
```

在程序的其他部分中，你将通过文件指针访问EMPS.DAT文件，而不是通过文件名访问。在每次访问文件时，使用文件指针变量比输入文件名和文件的完全路径更简单，而且更不容易出错。



警告　当你处理完文件后要关上档案柜的抽屉，否则会撞到你的头！在使用完文件后，要关闭所有打开的文件，否则你将丢失数据。fclose()与fopen()刚好相反。在它的括号里，fclose()要求写上想要关闭的文件的指针。



警告　如果文件指针等于0，你就知道发生了一个错误（比如硬盘驱动器的通道是打开的）。如果打开文件时发生了一个错误，C语言会从fclose()返回0。例如，如果你尝试打开一个不存在的硬盘文件，fclose()就会返回一个错误。

"w"（前面代码中fopen()的第2个参数）的意思是写（write）。fopen()的第2个参数必须是表28-1中的一个字符串模式（mode）值。

表28-1　基本的fopen()模式字符串

模　式	描　述
"w"	写（write）模式，无论文件是否存在都会创建一个新文件
"r"	读（read）模式，允许读一个已经存在的文件。如果文件不存在，就会产生一个错误
"a"	附加（append）模式，允许添加数据到文件的末尾，或者当文件不存在时创建文件

28.3　访问顺序文件

你只会对顺序文件做3件事：创建它、读它和向它添加数据（写它）。为了写文件，你可以使用fprintf()。fprintf()很容易使用，因为它只不过是在括号开始处带有文件指针的printf()。下面的程序创建一个文件并用fprintf()写入一些数据：

```
#include <stdio.h>
#include <stdlib.h>
FILE * fptr;
main()
{
  int age = 45;  /* Simple variables to write */
  float salary = 9670.50;
  fptr = fopen("C:\\MYDATA.DAT", "w"); /* Opens for output */
  if (fptr == 0)
    { printf("An error occurred while opening the file.\n");
      exit (1);
    }
  fprintf(fptr, "Here is some stuff:\n");
  fprintf(fptr, "I am %d years old.\n", age);
  fprintf(fptr, "I make $%.2f dollars every three months!/n",
          salary);
  fclose(fptr);  /* ALWAYS close your files */
  return 0;
}
```

如果你运行这个程序并查看名为MYDATA.DAT的文件内容（使用DOS的TYPE命令、文本编辑器或后面要介绍的C程序），你会看到：

```
Here is some stuff:
I am 45 years old.
I make $9670.50 dollars every three months!
```

用fgets()读取文件的内容。fgets()只不过是可以定向到硬盘文件的gets()。fgets()从文件中读取行到字符数组中（或者由一个字符型指针指向的已分配的堆内存）。



线索 把fputs()和fgets()开头处的f看成文件（file）。puts()和gets()分别到达屏幕和键盘，fputs()和fgets()从文件中写和读数据。

与gets()不同，fgets()要求指定要读入的数组的最大长度。如果不小心，你可能会读到超过文件结尾（产生一个错误），所以要确保检查文件结尾的位置。

fgets()一次读一行数据。如果在fgets()中指定的字符数比实际存在于你要读取的文件的行上的字符数更多，那么只要到达文件行末尾的换行符就停止读取数据。前面创建MYDATA.DAT文件的程序总是在每一行的结尾写上\n，这样接下来的fgets()函数就能够一行一行地读文件了。

下面的程序展示了如何读包含字符串数据的文件以及如何检查文件结束（用feof()函数）：

```
#include <stdio.h>
FILE * fptr;
main()
{
  char fileLine[81];  /* Will hold the output */
  fptr = fopen("C:\\MYDATA.DAT", "r");  /* Opens for input */
  if (fptr != 0)
    {  while (!feof(fptr))  /* Probably the only good
                               use for ! */
      { fgets(fileLine, 81, fptr);  /* Gets no more
                                       than 81 chars */
        if (!feof(fptr))
          { puts(fileLine); }
      }
    }
  else
    {  printf("\nError opening file.\n"); }

fclose(fptr);  /* ALWAYS close your files */
return 0;
}
```

如果你刚从文件中读取了最后一行，feof()就返回真。因为我们准确地知道MYDATA.DAT包含的内容（在更早的程序中刚刚创建了它），所以在前面的程序中其实不需要feof()。文件中只有3行，3个fgets()就足够了。但是，当读取硬盘文件时，一般要用feof()。你通常不能准确地知道文件中包含了多少数据，因为使用其他程序的其他人可能向文件中添加了数据。



请跳过这里，这有点儿难　在fprintf()函数中，文件指针是函数的第1个参数。在fget()函数中，文件指针是最后一个参数。看起来不太一致！



线索　如果你用相应的fprintf()把值写到数据文件中，那么也可以用fscanf()从中读取单个的数值。

可以通过附加模式打开文件来添加文件的数据以及把数据输出到文件中。下面的程序在MYDATA.DAT文件的最后添加一行That's all!：

```
#include <stdio.h>
FILE * fptr;
main()
{
  fptr = fopen("C:\\MYDATA.DAT", "a"); /* Opens for append */
  if (fptr == 0)
    { printf("An error occurred while opening the file.\n");
```

```
    exit (1);
  }
fprintf(fptr, "\nThat's all!\n");  /* Adds the line */
fclose(fptr);  /* ALWAYS close your files */
return 0;
}
```

下面是MYDATA.DAT现在的内容（注意额外的行）：

```
Here is some stuff:
I am 45 years old.
I make $9670.50 dollars every three months!
That's all!
```



线索 如果要写到打印机而不是屏幕，就要打开名为LPT1:或LPT2的DOS打印机设备，并用设备名作为fprintf()的第1个参数。



奖励

- 把长期数据存储在数据文件中。
- 在使用文件之前，用fopen()打开它。
- 使用完文件之后，总是要用fclose()关闭它。



陷阱

- 不要没有检查feof()就读文件，因为你可能上一次已经读到文件的最后一行。
- 一旦打开了文件，就不要使用文件名。要使用通过fopen()连接到文件的文件指针。
- 别忘了文件指针是fprintf()函数的第1个参数，是fputs()函数的最后一个参数。

28.4 小结

本章介绍了如何创建、读和写顺序文件。C程序必须在打开文件之后，才能向它写入数据或者从它读取数据。当程序使用完一个文件，就应该关闭它。

当从文件读数据时，必须检查是否到达文件结尾以确保不会读取超出文件结尾的部分。feof()函数是用来检查文件结尾的内置C语言函数。

1. 代码范例

```
#include <stdio.h>
FILE * tvFile;
main()
{
  char name[] = "Gerald";
  char show[] = "Sign Field";
  int age = 18;
  tvFile = fopen("C:\\SHOWS.DAT", "w");  /* Opens for
                                            output */
  if (tvFile)  /* Same as tvFile != 0) */
    { fprintf(tvFile, "My name is %s.\n", name);
      fprintf(tvFile, "My favorite TV show is %s.\n", show);
      fprintf(tvFile, "I am %d years old.", age);
      fclose(tvFile);
    }
  else
    { printf("There's a problem with opening the file.\n"); }
  return 0;
}
```

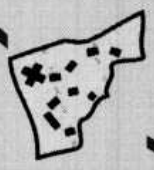
2. 代码分析

这段代码是把几条消息存储到一个顺序文件中的完整程序。首先定义了文件指针，然后在变量中存储了一些数据。

随后打开了一个文件。如果打开文件时没有错误发生，那么在文件关闭之前有3条消息被写到硬盘上。

这个程序很简单，但却是你编写更复杂的数据库程序的开端。你可能想要求用户输入一些值并把用户输入的数据存储起来，而不是直接把数据赋给变量。

29



值是大写的
258



fseek()使你能定位
文件指针
259



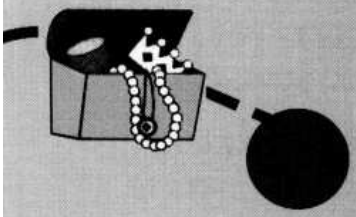
针对随机访问的文件
使用fseek()
258



访问模式是字符串
257



文件使用完要关闭它
260



第29章

另一种保存文件的方式

随机文件

本章内容

打开随机文件
256

访问随机文件
257

本章将讲解如何在文件中跳跃式地读取和写入数据。上一章介绍了一些写、读或附加数据到文件的方法。问题是一旦你以读模式打开一个顺序文件，就只能读它了。

有时候你会想从硬盘上读取一个客户的结构并修改他的余额。你当然不想只因为这一个修改就创建一个新文件。你会想把客户信息读到变量中，修改它，然后把它写回原来所在的硬盘位置上。如图29-1所示，随机文件使你可以在文件中跳跃式地读取和写入数据。

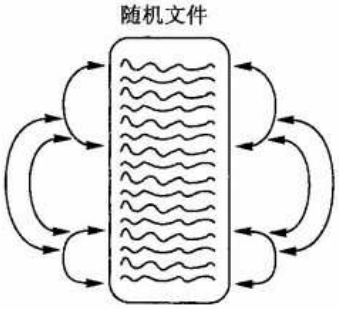


图29-1　随机文件使你可以按任何顺序读取和写入数据

文件的物理布局不能决定文件的类型（是随机还是顺序）。可以顺序地创建文件然后随机地读取并修改它。对C语言而言，文件只是一个字节流，而访问它的方式与文件的格式无关。

29.1　打开随机文件

为了随机地读或写文件，必须随机地打开文件。表29-1列出了访问随机文件的模式。如你所见，随机访问文件的基础是把加号和在上一章中学习的访问模式组合使用。



说明　与顺序文件一样，访问模式是作为fopen()的最后一个参数的字符串。可以用fclose()关闭随机文件，正如顺序文件一样。

表29-1　随机访问的fopen()模式

模　式	描　述
"r+"	打开一个已经存在的文件以供读和写
"w+"	打开一个新文件以供读和写
"a+"	以附加模式打开一个文件（文件指针指向文件的结尾），但允许回到文件的前面进行读和写

所有的3种模式都允许读和写文件。所选择的访问模式取决于你首先要对文件做什么。如果文件存在，而且你想随机地访问文件，就用r+模式。如果想创建文件，就用w+。如果文件已经存在，C语言会覆盖已存在的版本。如果想向文件的末尾添加数据，但也可能“往回”读或写已经存在的数据，就用a+。

下面是一条简单的fopen()语句，它打开一个已存在的文件以供写和读：

```
fptr = fopen("C:\\LETTERS.DAT", "w+");  /* Opens for
                                           write, then read */
```

与顺序文件一样，fptr变量必须是文件指针变量。如果你指定路径名，就要用双反斜杠。请记住如果打开失败，fopen()就返回0。



说明 如果你愿意可以指定一个长的路径名。例如，文件名可能像这样：

```
"C:\\ANDY\\WORK\\CORRESP\\LETTERS.DAT"
```

另外，可以把文件名存储在字符数组中，并用字符数组名代替实际的文件名的字符串字面量。文件名可以是大写字母或小写字母。

29.2 访问随机文件

趣闻轶事
信用卡公司在随机访问的文件中保存购买记录。

使用fseek()函数在文件中移动。在打开文件之后，C语言把文件指针初始化为指向你可以读或写的下一个位置。fseek()移动文件指针，使你可以在使用顺序访问时通常不能指向的位置进行读或写。下面是fseek()的格式：

```
fseek(filePtr, longVal, origin);
```

*filePtr*是用在使用随机访问模式的fopen()函数中的文件指针。*longVal*是一个long int型变量或字面量，可以是正数或负数。*longVal*是在文件中向前或向后移动的字节数。*origin*是表29-2所列的值之一。*origin*告诉fseek()从哪里开始移动。

表29-2 fseek()中出现的*orgin*的值

Origin	描 述
SEEK_SET	文件的开头
SEEK_CUR	文件的当前位置
SEEK_END	文件的结尾

*origin*值告诉C语言下一次你想从哪里开始访问随机文件。一旦你用fseek()指定了文件指针的位置，就可以用文件输入和输出函数来向文件写数据或从文件读数据。如果你指定的文件指针位置是文件末尾（使用SEEK_END），然后写入数据，那么新的数据将会增加到文件的末尾。如果你指定的文件指针位置在已有的数据上（使用SEEK_SET和SEEK_CUR），然后写入新的数据，那么新的数据将取代原有的数据。



警告　fseek()只能用于随机访问的文件。顺序文件只能按照数据的顺序访问。



线索　表29-2中的值是大写的，这表明了它们在某处被定义。在STDIO.H头文件中用#define指令定义了它们。

下面的程序以随机访问的模式打开一个文件，把字母A到Z写到文件中，然后重新读回这些字母。由于采用了随机访问模式w+，所以在开始读之前不必重新打开文件。

```
#include <stdio.h>
#include <stdlib.h>
FILE * fptr;
main()
{
  char letter;
  int i;
  fptr = fopen("C:\\LETTERS.DAT", "w+");  /* Opens for
                                           write, then read */
  if (fptr == 0)
    { printf("An error occurred while opening the file.\n");
      exit(1);
    }
  for (letter = 'A'; letter <= 'Z'; letter++)
    { fputc(letter, fptr);
    }
  puts("Just wrote the letters A through Z");
  /* Now reads the file backwards */
  fseek(fptr, -1, SEEK_END);  /* Minus 1 byte from the end */
  printf("Here is the file backwards:\n");
  for (i = 26; i > 0; i--)
    { letter = fgetc(fptr);
      fseek(fptr, -2, SEEK_CUR);  /* Reads a letter, then
                                     backs up 2 */
      printf("The next letter is %c\n", letter);
    }
```

```
    fclose(fptr);
    return 0;
}
```



线索 如你所见，fputc()是把单个字符输出到文件中的主要函数。fgetc()从文件中读取单个字符。fputc()和fgetc()与putc()和getc()的关系就像fputs()和fgets()与puts()和gets()的关系一样。



请跳过这里，这有点儿难 到目前为止，你可能还没有看到随机访问文件的目的。然而，随机访问文件使你可以不需要关闭并打开文件，就能向文件中写入数据并重新读取相同的数据。另外，fseek()使你能够把文件指针指定在从文件开头、文件中间或文件结尾开始计算的任意字节数的位置上。

假设在前面两个程序运行之后，字母文件仍然存在于硬盘上，下一个程序询问用户他想修改哪个位置。程序随后用fseek()指定文件指针的位置并在那里写入一个*，然后用fseek()回到文件的开头并再次打印它。

```
#include <stdio.h>
#include <stdlib.h>
FILE * fptr;
main()
{
  char letter;
  int i;
  fptr = fopen("C:\\LETTERS.DAT", "r+");  /* Opens for read,
                                             then write */
  if (fptr == 0)
    { printf("An error occurred while opening the file.\n");
      exit(1);
    }
  printf("What is the position you want to change? ");
  scanf(" %d", &i);
  /* Seeks to that location from the beginning of the file */
  fseek(fptr, (i - 1), SEEK_SET);  /* Subtracts 1 because
                                      first position is 0 */
  /* Writes '*' over that position */
  fputc('*', fptr);
  /* Now prints the file */
  fseek(fptr, 0, SEEK_SET);  /* Back to the beginning */
  printf("Here is the file now:\n");
  for (i = 0; i < 26; i++)
    { letter = fgetc(fptr);
      printf("The next letter is %c\n", letter);
    }
```

```
    fclose(fptr);
    return 0;
}
```

程序在用户指定的位置写入*之后，输出文件的内容。示例如下：

```
What is the position you want to change? 4
Here is the file now:
The next letter is A
The next letter is B
The next letter is C
The next letter is *
The next letter is E
The next letter is F
The next letter is G
The next letter is H
The next letter is I
The next letter is J
  :
  :
```

如你所见，按字母顺序的文件的第4个位置（字母D）现在是一个星号。文件的其他部分没有修改。



奖励

- 如果需要修改文件中的数据，就在fopen()模式字符串中使用加号。
- 请记住fseek()使文件指针在随机文件中移动，使你能够从开头、中间或结尾开始读或写。



陷阱

- 当你用完文件之后，不要忘了关闭它。
- 如果fopen()打开文件失败了（返回0），就不要试图使用这个文件了。

29.3　小结

本章讲解了随机访问的文件是如何工作的。一旦你用随机访问模式打开了一个文件，就能够按照所需要的任何顺序读或写文件了。fseek()是用来从文件的一个位置跳到另一位置的内置函数。

当你想更新文件的数据时，能够修改文件的内容是很重要的。你经常会希望在不重写整个文件的情况下修改一个人的地址以及修改存货的数量等。但如果你使用

顺序文件，就不得不修改整个文件。

1. 代码范例

```
#include <stdio.h>
#include <stdlib.h>
  struct std {
    char id[4];  /* 3-character ID plus a null zero */
    float avg;
  };
FILE *gradePtr;
main()
{
  struct std students;  /* Student records */
  gradePtr = fopen("C:\\GRADES.DAT", "w+");
  if (gradePtr == 0)
    { printf("An error occurred while opening the file.\n");
      exit(1);
    }
  do {
    printf("What is the next student's 3-character ID?\n");
    printf("Enter Q to quit entering students): ");
    gets(students.id);
    if (students.id[0] != 'Q')
      { printf("What is the student's average? ");
        scanf(" %f", &students.avg);
        getchar();  /* Gets rid of the Enter keypress */
        fprintf(gradePtr, "%s%.1f\n", students.id,
                students.avg);
      }
    } while (students.id[0] != 'Q');
  fclose(gradePtr);
  return 0;
}
```

2. 代码分析

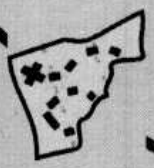
这段代码是一个把结构数据写入到随机访问的文件的完整程序。`fprintf()`确保了每个结构都按相同的格式和长度——包含3个字符的字符串后面跟上一个浮点值——写入硬盘。

可以很容易地用`fseek()`编写代码来读取或修改文件中的任何记录：

```
fseek(gradePtr, recNo * sizeof(struct std), SEEK_SET);
```

其中的`recNo`是用户想修改的记录数。

30



变量g2是全局的
268



独立的函数能使
你专注于代码
264



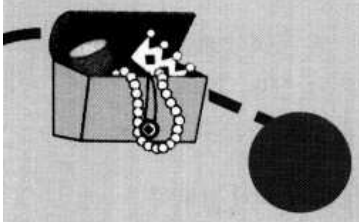
如果只用
局部变量
268



每个函数就
像一块积木
265



不要一开始学习
就使用全局变量
269



第30章

用函数来组织程序

使用函数

本章内容

用C函数来组织程序
264

局部变量和全局变量
267

一般的计算机程序不是像你在教科书上看到的只有20～30行。在“现实世界”中，计算机程序要长得多，但是长的程序包含很多妨碍学习新概念的代码。这就是为什么直到现在，你看到的都是相当短的程序，它们的代码都写在main()函数中。

如果main()函数中的程序很长，将来如果需要修改它，就要花费很多时间来寻找特定的代码。从本章开始的这3章将探索使用多个函数来划分程序的方法。通过分段对代码分类使程序更容易编写和维护。



请跳过这里，这有点儿难　人们得编写、修改以及调整程序。通过编写许多完成单项任务的函数使程序越清晰，你就能更加轻松高效地完成编程任务！你将会明白，分开的函数能使你把精力集中于需要修改的代码上。

30.1　用C函数来组织程序

你可能知道，C程序不同于BASIC程序。C程序被设计成迫使你按照模块化的思想来考虑函数的使用。BASIC程序通常只有一个非常长的程序，不会划分成不同的过程。BASIC的一些新版本，如QBASIC，现在提供的功能与C语言所提供的几乎相同。C程序不是只有一个很长的程序。它由许多称为函数（function）的程序构成。其中一个函数（总是需要的并且通常放在最前面的那个）叫做main()函数。

如果程序做许多事情，就应把它划分成几个函数。每一个程序应该完成一个主要任务。例如，如果你正在编写一个C程序要实现从键盘上获取一个数字列表，并对其排序，然后打印到屏幕上，那么你可以把所有工作写在一个大的函数中——所有的工作都在main()函数中——如下面的程序大纲所示：

```
main()
{
  /* This is not a working program, just an outline */
     :
  /* C code to retrieve a list of numbers */
     :
  /* C code to sort the numbers */
     :
  /* C code to print the sorted list on-screen */
     :
  return 0;
}
```

趣闻轶事
把程序划分成更小的函数叫做结构化编程。

这个程序没有提供一个好的格式，因为它太连续了。尽管有几项不同的任务要做，但是所有的代码都写在main()函数中。这个程序可能不需要很多行代码，但是最好养成把程序划分成不同的任务的习惯。

不要用main()来做所有的事。实际上，main()除了调用其他函数之外不应该做太多事情。组织这个程序的一个更好的方法是为程序要完成的每项任务编写单独的函数。



说明 当然，每个函数不应该只有一行，但是要确保每个函数就像一块积木并且只完成一个任务。

下面是一个更好的程序大纲：

```
main()
{
  getNums();     /* Calls a function that gets the numbers */
  sortNums();    /* Sorts the numbers */
  printNums();   /* Prints the numbers on-screen */
  return 0;      /* Quits the program */
}
/* A second function */
getNums()
{   :
  /* C code to retrieve a list of numbers */
  return;
}
/* A third function */
sortNums()
{   :
  /* C code to sort the numbers */
  return;
}
/* A fourth function */
printNums()
{   :
  /* C code to print the sorted list on-screen */
  return;
}
```



说明 尽管这个程序大纲比前一个更长，但是这个大纲组织地更好，而且正因为如此，它更容易维护。main()所做的唯一事情就是控制其他函数，即通过调用其他函数来实现。

每一个函数都完成自己的任务然后返回到main()中，main()再调用其他函数直到程序结束。然后main()返回到操作系统中。main()的功能就像一个程序的目录表。加上恰当的注释，main()能使你准确地知道需要修改的代码在哪个函数中。



线索　有一个经验性的好规则：一个函数的代码行不应该超出一个屏幕。如果超出了，这个函数很可能做了太多的事情。在中学时代，难道你会不讨厌阅读章节很长的书吗？同样你也不会喜欢使用函数很长的程序。

任何函数都能调用另一个函数。例如，如果想让printfNums()在一页的顶部打印带有你的名字和日期的标题，你可能要让printfNums()调用另一个叫做printfTitle()的函数。printfTitle()执行完后会返回到printfNums()中。下面是代码的主体内容：

```
main()
{
  getNums();     /* Calls a function that gets the numbers */
  sortNums();    /* Sorts the numbers */
  printNums();   /* Prints a title AND the numbers on-screen */
  return 0;      /* Quits the program */
}
/* A second function */
getNums()
{   :
  /* C code to retrieve a list of numbers */
  return;
}
/* A third function */
sortNums()
{   :
  /* C code to sort the numbers */
  return;
}
/* A fourth function */
printNums()
{   :
  /* C code to print the sorted list on-screen */
  printTitle();
  return;
}
/* A fifth function called from printNums() */
printTitle()
{
  /* C code to print a title */
  return;  /* Returns to printNums(), not to main() */
}
```



说明 查看附录B中21点游戏里的所有函数。程序只有几页，但它包含了数个函数。查阅代码并且看看是否能找到一个函数，它调用了另一个位于程序中其他位置的函数。

整个电子学产业也从编程中学习了一些东西。现今的大多数电子元件，如电视、电脑和收音机，包含了许多电路板，它们可以移除、更新和替换，而不会影响系统的其他部分。类似地，要能够修改程序的某一部分：如果通过使用函数编写了一个组织良好的程序，那么就可以只修改需要修改的函数，而不必处理许多不相关的代码。

30.2 局部变量和全局变量

前一节介绍的程序大纲还需要更多的代码才能使用。在添加代码之前，需要进一步弄清变量的定义。在C语言中，所有的变量可以是局部的（local）或全局的（global）。到目前为止，你所见过的所有变量都是局部的。大多数时候，局部变量比全局变量更安全，因为局部变量提供的是一种需要知道的访问（need-to-know access）。也就是说，如果一个函数需要一个变量，那么它可以通过变量传递过程（下一章会介绍）来访问另一个函数中的局部变量。

如果一个函数不需要访问另一个函数的局部变量，它就没有访问权限。任何函数都可以对全局变量进行读取、修改和清零操作，所以全局变量的安全性较差。

下面的规则描述了局部变量和全局变量的区别：

- 一个变量是全局的当且仅当变量在函数名的前面定义（如`int i;`）；
- 一个变量是局部的当且仅当变量在一个左大括号之后定义。函数总是以左大括号开始。一些语句，如`while`，也有左大括号，也可以在这些括号中定义局部变量。



线索 一对左大括号和右大括号之间的内容称为块（block）。

按照这些规则，在下面的程序中很明显`l1`和`l2`是局部变量，而`g1`和`g2`是全局变量。

```
#include <stdio.h>
int g1 = 10;
main()
{
  float l1;
  l1 = 9.0;
  printf("%d %f\n", g1, l1); /* Prints global g1 and local */
  prAgain();                 /* l1, calls next function,   */
  return 0;                  /* and returns to DOS         */
}
float g2 = 9.0;  /* A global variable */
/* Next is a separate function */
prAgain()
{
  int l2 = 5;
  printf("%d %f %d\n", l2, g2, g1);  /* Can't print l1! */
  return;
}
```



请跳过这里，这有点儿难　你可能还没有理解return 0;语句。更糟的是，在prAgain()函数的末尾出现了一个单独的return。你将在接下来的两章中找到return的详细描述。



线索　变量g2是全局的，因为它定义在函数prAgain()之前。

局部变量只能在它们自己的代码块中使用。因此，l1不能在prAgain()中打印或修改，因为l1是main()的局部变量。相反地，l2不能在main()中使用，因为l2只能在prAgain()中可见。变量g1是整个程序可见的，变量g2只在其定义之后的程序中可见。



线索　所有的全局变量在源文件中的定义点之后有效。不要在程序的中间定义全局变量（就像前一个程序那样）因为在调试时会很难找到它的定义。应该限制（或消除）使用全局变量。如果一定要使用全局变量，就把它们全部定义在main()函数的前面，这样容易找到（比如如果你需要修改它们或者查看定义的数据类型）。



警告　前面展示的程序框架有一个问题。如果只使用局部变量（而且应该总是尝试这样做），那么在getNums()中输入的变量值就既不能在sortNums()中排序也不能在printNums()中打印了！请稍等，下一章将解决这个问题。



请跳过这里，这有点儿难 如果编译前面的程序，编译器会给出一个关于调用没有原型函数的警告，现在先暂时忽略这个错误。这个问题在第32章中将得到解答。



奖励

- 在块的左大括号之后定义局部变量。在函数开始之前定义全局变量。
- 局部变量比全局变量更安全，所以尽可能地使用局部变量。
- 把程序划分为许多函数会使维护更容易，并且也会加快程序开发。



陷阱

- 不要在程序的中间定义全局变量。如果这样做，会很难找到它们。
- 不要一开始就使用全局变量。随着程序的编写，会偶尔遇到需要全局变量的时候，这时再加上（下一章将讲解更多关于用局部变量而不用全局变量的内容）。

30.3 小结

本章教你用搭积木的方法来编写C程序。长的程序是很难使用的，除非把它们划分成单独的函数。一个长的main()函数就像一本很长的没有划分章节的书。要把长的程序划分成单独的函数并且让每一个函数在程序中执行单一的、不同的任务。

一旦把程序划分成几个函数，就得考虑如何在代码中使用变量了。局部变量定义在函数内部并且只能在该函数中使用。与局部变量相反的是全局变量，它的值可以在位于其定义点之后的所有函数中使用。不赞成使用全局变量。局部变量更安全，因为可以限制只有那些需要使用它们的函数才能访问它们。在下一章中，你将学到如何在函数之间共享局部变量。

1. 代码范例

```
#include <stdio.h>
char name[] = "Mary";   /* Global */
main()
{
  int age = 25;  /* Local variable */
```

```
  /* The following printf() can use name
     because name is global */
  printf("%s is %d years old.\n", name, age);
  nextFun();  /* Calls second function */
  return 0;   /* and returns to DOS    */
}

float weight = 119.0;  /* A global variable */
/*****************************************************/
nextFun()
{
  /* The next printf() CANNOT print age because
     age is local and usable only in main() */
  printf("%s is %d years old and weighs %.0f pounds.\n",
         name, age, weight);
  return;  /* Goes back and finishes main() */
}
```

2. 代码分析

这个程序包含了3个变量，其中两个是全局的（name和weight），另一个是main()的局部变量（age）。第1个函数main()，在调用nextFun()时被暂停执行剩下的代码。当nextFun()执行完后，main()的最后一行才会继续执行。

nextFun()函数不能使用age的值，因为age是main()的局部变量，因此只能在main()中使用。

这个程序在nextFun()函数前面有一行全部是星号的注释。程序员通常使用这样的注释来区分不同的函数。当查阅代码时，可以更容易地找到想要的函数。

31



那些空括号有用处
275



按地址传递非数组变量
278



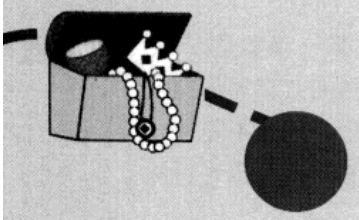
按值传递保护了变量
276



传递的值是重要的
275



不要按值传递数组变量
280



第 31 章

在函数间共享数据

通过传递变量

本章内容

传递参数
274

传递实参的方法
274

上一章留下了一些未解决的问题。如果使用多个函数是好习惯（确实是），而且如果使用局部变量是好习惯（确实是），那么必须有方法在函数间共享局部变量才完美（确实有办法）。不会想让所有的函数都能访问所有的变量，因为不是每一个函数都需要访问每一个变量。如果在函数间需要完全的变量访问，那么也许可以使用全局变量。

为了在不同的函数间共享数据，必须把变量从一个函数传递（pass）到另一个函数。当一个函数把一个变量传递到另一个函数时，只有这两个函数能访问此变量（假设此变量是局部的）。本章讲解如何在函数间传递变量。

31.1　传递参数

当从一个函数传递变量到另一个函数时，你是在传递一个实参（passing an argument）。一次可以传递多个变量。接收者函数从发送变量的函数接收形参（receives the parameters）。



警告　单词变量（variable）、实参（argument）和形参（parameter）在传递和接收数值时可以相互交换使用。名称不重要，重要的是理解发生了什么。图31-1帮助解释这些术语。

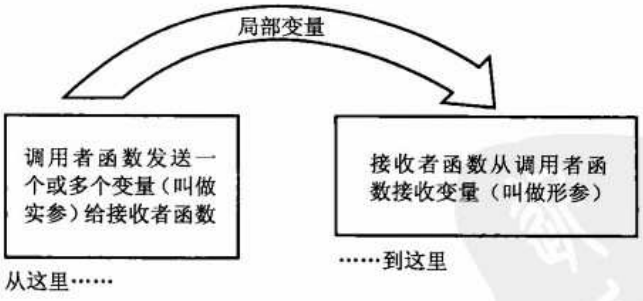


图31-1　正确使用术语

31.2　传递实参的方法

有两种传递实参的方法：按值（by value）和按地址（by address）。这两种方法都从调用者函数传递实参到接收者函数。还有一种办法可以从一个函数返回

（return）值到调用者函数中（参见下一章）。



线索 所有传递数值的讨论都集中于函数名后面的括号。是的，空的括号毕竟也有用！想要传递的变量放在函数调用的括号中，也出现在接收者函数中，在下一章中将会看到。



说明 是的，传递数值这部分内容很重要！不过，你将发现它其实很容易。

31.2.1 按值传递

有时按值传递（passing by value）被叫做按副本传递（passing by copy）。这些术语可相互交换使用，因为它们意味着同一件事。按值传递意味着变量的值（value）被传递到接收者函数，而不是变量本身。下面的程序从main()函数传递一个值到half()函数：

```
#include <stdio.h>
main()
{
  int i;
  printf("Please enter a number... ");
  scanf(" %d", &i);
  /* Now, passes the variable to a function */
  half(i);  /* i is passed by value */
  printf("In main(), i is still %d.\n", i);
  return 0;  /* Goes back to DOS */
}
/*********************************************/
half(int i)  /* Receives value of i */
{
  i = i / 2;  /* Halves i */
  printf("Your value halved is %d.\n", i);
  return;  /* Returns to main() */
}
```

下面是程序的输出：

```
Please enter a number... 12
Your value halved is 6.
In main(), i is still 12.
```

分析一下half()函数的第1行：

```
half(int i)  /* Receives value of i */
```

请注意必须把数据类型（int）写在接收者函数的形参列表中。如图31-2所示，i的内容被传递给half()函数。main()函数中的i不会被修改，因为只是传递了i值的一个副本。

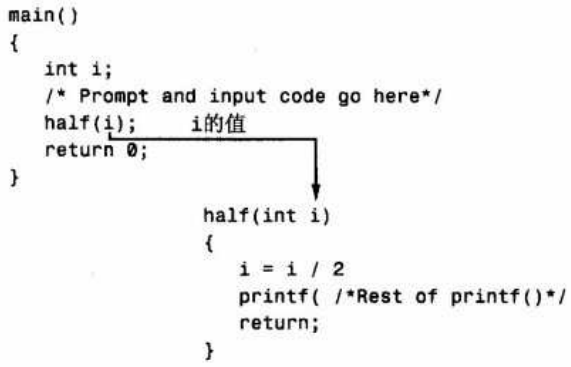


图31-2　传递了i的值，而不是变量i本身

如果传递了由逗号分隔的多个参数，即使它们的类型都相同，也要列出所有的数据类型。下面的函数接收3个变量：一个浮点值、一个字符数组和一个整数：

```
aFun(float x, char name[15], int age)   /* Receives three
arguments */
```



警告　按值传递保护了变量。如果接收者函数修改了一个按值传递的变量，那么传递者函数中的变量并不会改变。因此，按值传递总是安全的，因为接收者函数无法修改传递者函数的变量——只能使用它们。



说明　如果前面程序中的接收者函数的参数名改为i2，那么程序仍然会按照现在的方式运行。i2是half()函数中的局部变量，且与main()函数无关而main()中的i是main()函数中的局部变量。

C语言对所有的非数组变量使用按值传递。因此，如果传递不是数组的任何变量给函数，那么只传递了变量值的一个副本。无论接收者函数对值进行什么操作，传递者函数中的变量都保持不变。

31.2.2　按地址传递

当传递数组给另一个函数时，数组是按地址传递的。不是传递数组的副本，而

是传递数组的内存地址。接收者函数随后把它的接收形参数组放在传递来的地址上。关键是接收者函数与传递者函数工作在相同的地址上。如果接收者函数修改了形参列表中的一个变量，那么传递者函数的实参也修改了。

下面的程序传递一个数组给函数。函数把x赋给数组的每个元素，然后main()打印数组。请注意main()打印出所有的x，因为函数修改了实参。

```
#include <stdio.h>
#include <string.h>
main()
{
  char name[15] = "Chris Williams";
  change(name);
  printf("Back in main(), the name is now %s.\n", name);
  return 0;
}
/*******************************************************/
change(char name[15])
{
  /* Change the string stored at the
     address pointed to by name */
  strcpy(name, "xxxxxxxxxxxxxx");
  return;
 }
```

程序产生下面的输出：

```
Back in main(), the name is now xxxxxxxxxxxxxx.
```

如果不想按值传递非数组变量，可以强迫C语言按地址传递普通的非数组变量。然而，这样做看起来有点疯狂！下面的程序与本章的第1个程序很类似，但是会产生不同的输出：

```
#include <stdio.h>
main()
{
  int i;
  printf("Please enter a number... ");
  scanf(" %d", &i);
  /* Now, passes the variable to a function */
  half(&i);  /* i is passed by address */
  printf("In main(), i is now %d.\n", i);
  return 0;  /* Goes back to DOS */
}
/*****************************************/
half(int *i)  /* Receives address of i */
{
  *i = *i / 2;  /* Halves i */
  printf("Your value halved is %d.\n", *i);
  return;  /* Returns to main() */
}
```

下面是程序的输出：

```
Please enter a number... 12
Your value halved is 6.
In main(), i is now 6.
```



请跳过这里，这有点儿难　这看起来有点怪，但是如果想按地址传递非数组变量，就要在传递函数中在变量前面加上&（取地址）符号，并在接收者函数中变量每一次出现的前面加上*（解引用）符号。如果你认为现在是传递指针给函数，那么就对了。



线索　现在不难理解scanf()了。记得在传递给scanf()的非数组变量前面要加上&，而数组变量则不用。当调用scanf()时，必须传递变量的地址以便于修改变量。因为字符串是数组，所以当从键盘上获取字符串时，不必在数组名前面加上取地址运算符。

下面的程序按值传递整数i，按地址传递浮点数x，并按地址传递一个整型数组（所有的数组都这样传递）：

```
#include <stdio.h>
/* You'll understand the next statement
   after the next chapter! */
changeSome(int i, float *newX, int iAry[4]);

main()
{
  int i = 10;
  float x = 20.5;
  int iAry[] = {10, 20, 30, 40, 50};
  puts("Here are main()'s variables before the function:");
  printf("i is %d\n", i);
  printf("x is %.1f\n", x);
  printf("iAry[0] is %d\n", iAry[0]);
  printf("iAry[1] is %d\n", iAry[1]);
  printf("iAry[2] is %d\n", iAry[2]);
  printf("iAry[3] is %d\n", iAry[3]);
  printf("iAry[4] is %d\n", iAry[4]);
  /* Without the &, x would pass by value */
  changeSome(i, &x, iAry);
  puts("\nmain()'s variables after the function:");
  printf("i is %d\n", i);
  printf("x is %.1f\n", x);
  printf("iAry[0] is %d\n", iAry[0]);
```

```
  printf("iAry[1] is %d\n", iAry[1]);
  printf("iAry[2] is %d\n", iAry[2]);
  printf("iAry[3] is %d\n", iAry[3]);
  printf("iAry[4] is %d\n", iAry[4]);
  return;
}
/************************************************/
changeSome(int i, float *newX, int iAry[4])
{
  /* Change all parameters, but only the last
     two remain changed once main() returns */
  i = 47;
  *newX = 97.6;  /* Same location as x */
  iAry[0] = 100;
  iAry[1] = 200;
  iAry[2] = 300;
  iAry[3] = 400;
  iAry[4] = 500;
  return;
}
```

下面是程序的输出：

```
Here are main()'s variables before the function:
i is 10
x is 20.5
iAry[0] is 10
iAry[1] is 20
iAry[2] is 30
iAry[3] is 40
iAry[4] is 50

main()'s variables after the function:
i is 10
x is 97.6
iAry[0] is 100
iAry[1] is 200
iAry[2] is 300
iAry[3] is 400
iAry[4] is 500
```

下一章讲解如何从一个函数返回一个值到另一个函数，从而结束关于函数间值的传递的讨论。另外，最后你将理解STDIO.H的真正用途。



奖励

- 如果想让函数共享局部变量，就从一个函数传递局部变量给另一个函数。
- 如果想保护变量的值，就按值传递变量。
- 如果想让变量的值被修改，就按地址传递变量。
- 在想要按地址传递的非数组变量前面加上&。如果想传递数组，就不要加&。



陷阱

- 不要按值传递数组变量。在C语言中无法做到这一点。

31.3　小结

本章讲解了如何在函数间共享局部变量。当一个函数需要访问定义在另一个函数中的局部变量时，必须传递变量。函数名后面的括号里包含了传递和接收的变量。

通常按值传递非数组变量，这意味着接收者函数可以使用它们，但不会影响传递者函数中的值。数组是按地址传递的，这意味着如果接收者函数修改了它们，那么传递者函数中的数组变量也修改了。可以按地址传递非数组变量，只要在它们前面加上取地址运算符（&），然后用解引用运算符（*）来接收它们。

1. 代码范例

```
#include <stdio.h>
main()
{
  int i, j, k;
  printf("I'll add three variables. \n");
  printf("Please enter the first number: ");
  scanf(" %d", &i);
  printf("Please enter the second number: ");
  scanf(" %d", &j);
  printf("Please enter the third number: ");
  scanf(" %d", &k);
  addThem(i, j, k);  /* Passes variables by value */
  return 0;  /* Goes back to DOS */
}
/*************************************************/
addThem(int i, int j, int k)
{
  int total;  /* Local to this function */
  total = i + j + k;
  printf("\nThe total of the three variables is %d.\n",
         total);
  return 0;  /* Returns to main() */
}
```

2. 代码分析

这个程序接收main()函数中的3个变量，然后在addThem()函数中打印它们的和。变量是按值传递给addThem()函数的。当函数接收变量时，必须在每个变量前面加上数据类型。addThem()函数的第1行不能写成这样：

```
addThem(int i, j, k)  /* Invalid. int is required
                         three times */
```

局部变量total存放3个被传递变量的和。在打印了和之后，控制就返回到main()函数中。本书的程序中写在return语句后面的0将在下一章中解释。



×32



这个程序用了long int数据类型
285



大多数编译器允许使用void类型
287



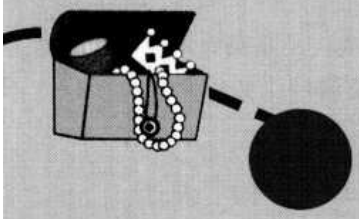
gets()函数实际上返回一个字符指针
285



原型不是必需的，但是强烈推荐使用
287



没有声明函数原型时，不要返回非整数
289



第 32 章

让函数更完美

使用返回值和函数原型

本章内容

返回值
284

返回的数据类型
286

声明函数原型
287

结语
289

本章不是学习C语言的结束。本章只是开始。听起来有点深奥，不是吗？本章说明如何从被调用的函数返回值到调用者函数中。另外，还要讲解函数的原型（prototype）。

关键内容是：你将理解为什么本书中的大多数程序包含`return 0;`一行，并且你将理解头文件的真正用途。

32.1　返回值

到目前为止已经学会如何发送变量给函数，也就做好了学习如何返回一个值的准备。当函数要返回一个值，使用`return`语句。通常，C程序员在`return`语句后面写上括号，在括号中写上返回的值，如`return (answer);`。



说明　如果函数没有返回值，就不需要`return`语句，因为函数会自动返回到调用者函数。但是，如果需要返回值，就要用`return`语句。

尽管可以传递多个实参给函数，但却只能返回一个值给调用者函数！图32-1解释了这一点。这条规则没有例外。

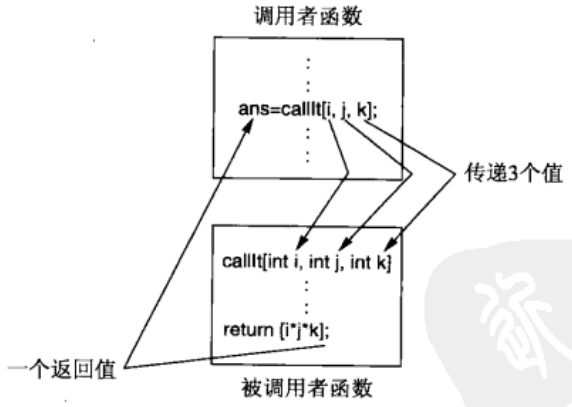


图32-1　可以传递多个值，但只能返回一个值

尽管只返回一个值可能看起来是有限制的，但其实不会。想一想内置的`sqrt()`函数。你也许还记得第20章中`sqrt()`函数返回传递给它的任何值的平方根。`sqrt()`函数没有返回许多值，只有一个。事实上，没有一个内置函数的返回值多于一个，自己写的函数也一样。



警告 gets()函数看起来似乎返回了多个值，因为它返回一个字符串数组。但是，请记住数组名只不过是数组的第1个元素的指针。因此，gets()函数实际上返回了一个字符指针，它指向用户输入的字符串的开头。

下面的程序包含了一个函数，它接收3个整型变量：a、b和c。名为fMul3()的函数把这3个值乘在一起并返回结果。

```
#include <stdio.h>
long int fMul3(int var1, int var2, int var3);
main()
{
  int a, b, c;
  long int answer;
  printf("What is the first number to multiply? ");
  scanf(" %d", &a);
  printf("What is the second number to multiply? ");
  scanf(" %d", &b);
  printf("What is the third number to multiply? ");
  scanf(" %d", &c);
  answer = fMul3(a, b, c);  /* Passes three variables and
                               returns the answer */
  printf("\nThose three values multiplied by each other "
         "equal %ld.", answer);
  /* It's okay to continue a string literal over two lines */
  return 0;
}
/*************************************************************/
long int fMul3(int var1, int var2, int var3)
{
  int locAnswer;
  locAnswer = var1 * var2 * var3;
  return (locAnswer);
}
```



线索 这个程序使用了long int数据类型，因为3个整数乘在一起可能会产生很大的数。

下面是这个程序的一个输出样例：

```
What is the first number to multiply? 3
What is the second number to multiply? 4
What is the third number to multiply? 5

Those three values multiplied by each other equal 60.
```



说明　请注意main()把fMul3()返回的值赋给answer。main()得对从fMul3()返回的值进行一些操作。

return后面可以跟表达式，也可以跟变量。下面的代码：

```
sales = quantity * price;
return (sales);
```

等价于：

```
return (quantity * price);
```

32.2　返回的数据类型

在fMul3()函数的开头，可以看到long int。long int是返回值locAnswer的数据类型。必须在任何有返回值的函数名前面写上返回的数据类型。如果函数返回一个float类型的值，那么float必须写在函数名的前面。

如果不指定返回的数据类型，C语言就默认为int。因此，C语言认为每一个没有显式指定返回的数据类型的函数将返回int。下面两个函数的第1行对C语言来说意味着同样的事：

```
int myFun(int a, float x, char c)
```

和

```
myFun(int a, float x, char c)  /* int is assumed */
```



线索　每一个main()函数都被认为返回一个int值，除非指定了返回的数据类型。这就是为什么在大多数程序的结尾看到return 0;！因为main()函数没有指定返回的数据类型，所以默认为int，而return 0;确保了一个int值被返回给DOS。顺便说一下，DOS通常会忽略返回的数据类型，除非你想用高级的DOS errorlevel命令来接收0。

如果函数没有返回值或者没有传递值，就可以在返回的数据类型或者形参列表上写上关键字void。因此，如果既不获取任何值也不返回任何值，函数的第1行会像下面这样：

```
void doSomething(void)  /* Neither is passed nor returns */
```



请跳过这里，这有点儿难 如果使用严格的ANSI C，main()函数的返回类型就不能是void类型。它必须是int类型。然而，大多数编译器，即使是那么把自己提升为兼容ANSI C的编译器，都允许指定void为main()函数的返回类型。

32.3 声明函数原型

趣闻轶事
飞行员经常在飞机原型中训练，它比真正的飞机提供了更安全的训练场所。

要使函数能够恰当地工作还差最后一步。如果函数返回不是int类型的任何值，就应该声明函数的原型。实际上，返回整数的函数也需要声明其原型。

单词原型的意思是某种事物的样式。函数的原型是实际函数的样式。起先，C语言中的函数原型声明看起来像是在浪费时间。

返回int类型值的函数不需要声明函数原型的原因是在没有指定不同返回值的情况下，int是默认函数原型返回值的类型。因此，下面的两个函数原型刻画了相同的函数：

```
int aFunc(int x, float y);  /* 2 passed, one integer returned */
```

和

```
aFunc(int x, float y);  /* 2 passed, one integer returned */
```



说明 如果函数没有返回值或者返回值是整型时，可以不需要声明函数原型，但是强烈建议加上它。一旦声明了原型，C语言就能确保不会把一个float类型值传给一个期待接收char类型值的函数。在没有声明函数原型的情况下，C语言会尝试将float转化为char，结果会传递一个错误的值。

为了声明函数的原型，要在main()之前的某处写上函数第1行的副本。在前面看到的程序中，fMul3()函数的原型正是出现在main()函数的前面。这一行不是函数调用，因为它出现在main()函数的前面。这一行也不是函数真正的第1行，因为函数原型的后面要加上分号。这一行是函数的原型。如果程序调用20个函数，就要有20个原型。

为程序中的每一个函数声明原型——每一个被代码调用的函数甚至是内置函

数，如printf()。你可能想知道当你没有在代码的前面写上printf()时，怎么就声明了它的原型。文件STDIO.H包含了printf()、scanf()、getchar()和其他许多输入输出函数的原型！strcpy()的原型在STRING.H头文件中。当学习了一个新的内置函数时，应该找到头文件的名字，以便于#include它并确保每个函数的原型都声明了。



请跳过这里，这有点儿难　main()函数不需要原型。只要把main()写在程序中的第1个函数，main()就不需要原型。main()被认为是无原型函数，因为在main()函数出现之前，没有其他函数叫做main()。

下面的程序不能正确地工作，因为float返回类型没有正确地声明其原型。请记住，除非在原型中指定了不同的返回类型，否则C语言假设返回的是一个int类型（即使返回的是另一种数据类型）。

```
#include <stdio.h>
compNet(float atomWeight, float factor);  /* Bad prototype */
main()
{
  float atomWeight, factor, netWeight;
  printf("What is the atomic weight? ");
  scanf(" %f", &atomWeight);
  printf("What is the factor? ");
  scanf(" %f", &factor);
  netWeight = compNet(atomWeight, factor);
  printf("The net weight is %.4f\n", netWeight);
  return 0;
}
/*********************************************************/
/* The function definition below doesn't work because the
   prototype does not indicate that a float is returned  */
compNet(float atomWeight, float factor)
{
  return ((atomWeight - factor) * .8);
}
```

下面是错误的输出：

```
What is the atomic weight? .0125
What is the factor? .98
The net weight is 0.0000
```

要修正错误，得把原型改为：

```
float compNet(float atomWeight, float factor);
```

并把conNet()的定义行（它的第1行）改成与原型匹配：

```
float compNet(float atomWeight, float factor)
```

32.4 结语

如果使用了全局变量，绝对不要传递或返回它。全局变量不需要传递。另外，在调用者函数、被调用者函数和原型中的参数列表在参数个数和数据类型上都应该相匹配（名字可以不必相同）。

现在你知道了关于传递参数和返回值的每一件事。进入正式程序员的思考状态并启动编译器吧！



奖励

- 在有返回值的函数名前面写上返回的数据类型。
- 返回值写在return语句后面。
- 在调用者函数中，可以将返回值输出，也可以将其赋给某个变量。如果不用返回值做一些事情，那么调用一个有返回值的函数是没有用处的。
- 如果函数既无返回值也不接收参数，就把void用在返回的数据类型和参数列表中。



陷阱

- 不要从一个函数返回多个值。
- 没有声明函数原型时，不要返回非整数。最好是为除了main()之外的所有函数声明其原型。

32.5 小结

本章通过讲解原型和返回值来完善关于函数的知识。当程序中包含许多函数时，在main()之前的某处声明这些函数的原型。这些函数原型让C程序知道所期待的是什么。在声明了原型之后，就可以传递和返回任何数据类型的变量。如果不声明函数原型，就只能返回int类型。

函数原型确保你不会因为不小必传递了错误的数据类型。例如，如果函数原型

表明应该传递两个float类型的值给一个函数，但是意外地传递了两个int类型的值，C程序就会报错。如果没有声明函数原型，C程序就不会报错，而且可能会因此而得到错误的结果。

既然知道了如何返回值，就可以编写实现内置函数（如sqrt()和rand()）功能的函数。调用函数时，函数会基于它的代码返回一个值。一个函数最多返回一个值，就像内置函数那样。

1. 代码范例

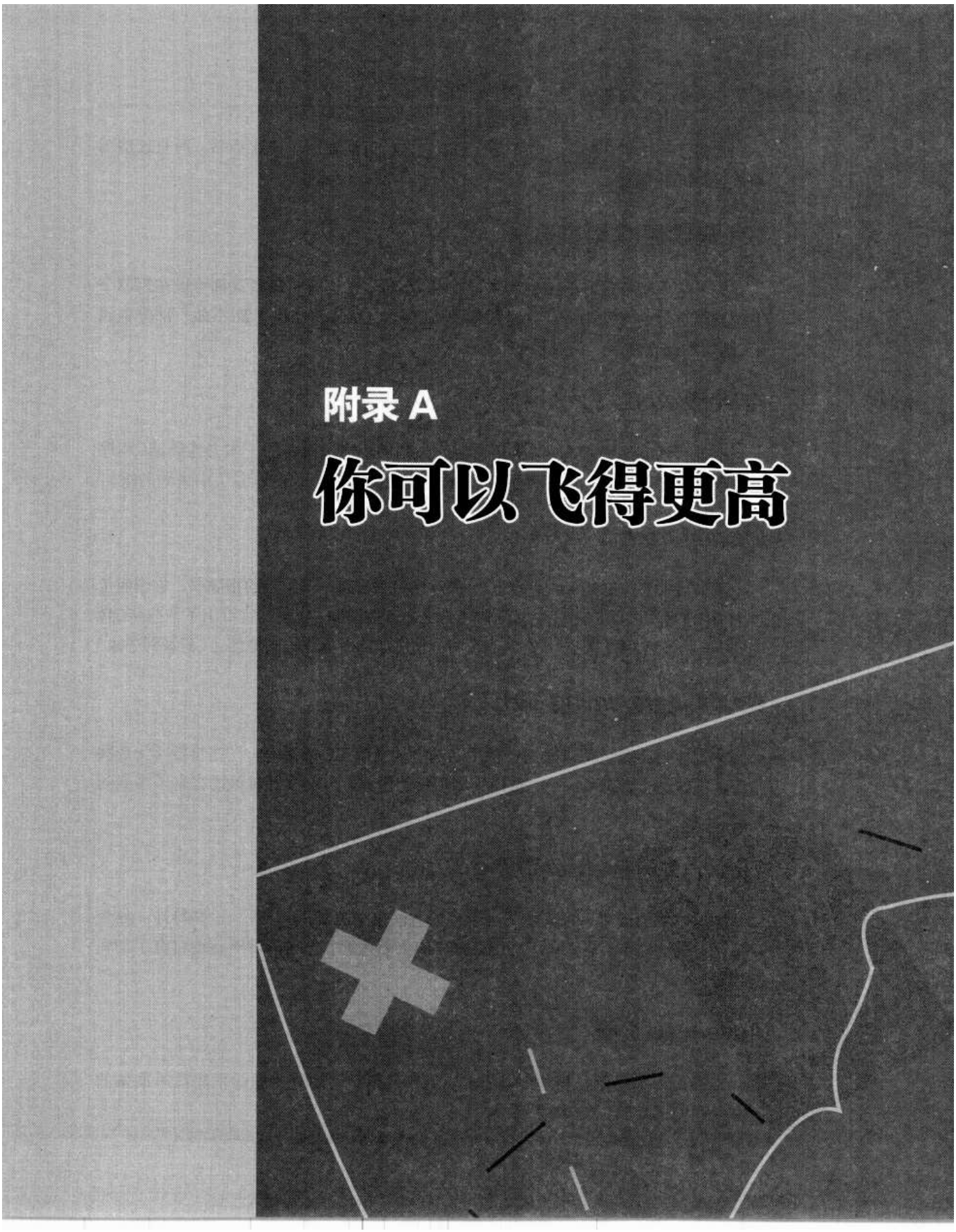
```
#include <stdio.h>
float divideIt(float numFromMain);  /* Prototype */
main()
{
  float myNum, half;
  printf("Please type a number: ");
  scanf(" %f", &myNum);
  half = divideIt(myNum);
  printf("Your number divided by two is %.1f.\n", half);
  return 0;
}
/*********************************************************/
float divideIt(float numFromMain)
{
  float halfNum;
  halfNum = numFromMain / 2.0;  /* Divides main()'s number */
  return (halfNum);  /* Sends the halved value back */
}
```

2. 代码分析

这个程序说明了如何声明原型，并调用了一个函数，此函数返回传递给它的值的二分之一。原型使C程序知道名为divideIt()的函数接收并返回一个浮点型的值。因为divideIt()函数的局部变量不能在main()函数中使用，所以它的值返回给main()函数并被main()函数中的变量half所捕获。

在main()函数中执行除2操作会更高效，但是这个例子更好地论证了本章的概念。下面是程序的一个输出示例，从中可以看出程序是如何工作的：

```
Please type a number: 45.6
Your number divided by two is 22.8.
```



附录 A

你可以飞得更高

本附录列出了一些书，在了解了C语言之后可能要读一读这些书。所有这些书都来自Sams出版社。

《21天学通C语言》

有了这本销量最好的书，用户可以成功地学会C语言！每堂课都可以在2到3个小时甚至更短的时间内完成。带阴影的语法框、Q&A部分和“做/不做”部分强调了C语言的重要主题。（初级到中级）

Advanced C

这是为那些想要提高自己C语言编程技巧的程序员准备的。本书为调试C程序以及改善程序的速度、内存使用和可读性给出了许多提示和技巧。（中级到高级）

Moving from C to C++

本书对于那些想要从C转变到C++的程序员来说是一本宝贵的指导书。它说明了一个用C语言写成的应用程序怎样转化为更加高效的C++代码。它给出了一些从C转变到C++的技巧，并在教学开始之前说明了使用面向对象编程的原因。（初级到中级）

C++ Programming 101

在这本书中读者可以采用一种主动的方法来学习C++语言。一些特性（如查找错误、试试这个、思考……、结束程序和仍然困惑？）使读者对该语言有了彻底的理解。（初级）

Turbo C++ Programming 101

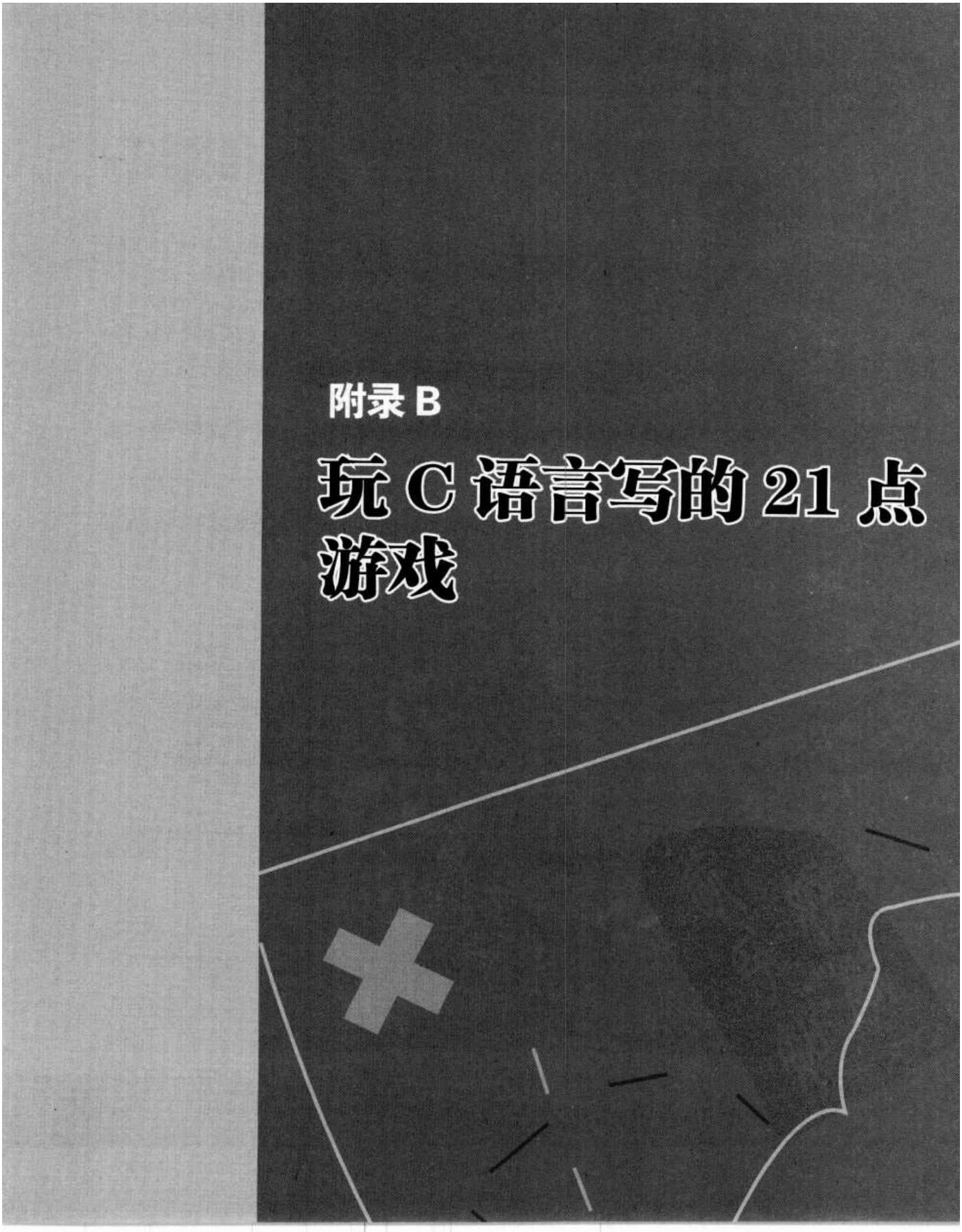
在这本书中读者可以采用一种主动的方法来学习Turbo C++。一些特性（如查找错误、试试这个、思考……、结束程序和仍然困惑？）使读者对该语言有了彻底的理解。（初级）

Advanced C++

本书是为那些已经精通C++基本知识并且想学习高级主题的程序员所准备的一本全面的指导书。（中级到高级）

附录 B

玩 C 语言写的 21 点游戏



编程并不只是枯燥的工作，它也可以变得很有趣，下面的21点游戏证明了这一点！此游戏提供了一个很长的例子，可以在学习C语言时研究它。尽管游戏写得十分简单，但程序中还是涉及了很多内容。

与所有编写良好的程序一样，这个程序有完备的注释。实际上，如果你已经读了本书的每一章，你将会理解21点游戏的编程。程序保持简单的原因之一是为了不受编译器的约束。例如，没有用到清屏函数。名为dispTitle()函数的注释解释如何添加一个比通常使用的清屏函数更加特别的清屏函数。另外，你可能想弄清你的C编译器是如何在屏幕上产生颜色以便于在游戏的显示上花一些精力。



线索　一旦掌握了足够的C语言并理解程序的内部工作，就可以开发图形功能并实际画出纸牌。

在许多代码行的左边有数字。这些数字是那些讨论本行所用概念的章号。如果对某行有疑问，就可以查看对应的章节。

```
1  /* Filename: BLAKJACK.C

3  This program plays a game of Blackjack with you. The
   computer is the dealer and you are the victim-er, I mean
   player. The dealer gets a card that you can see. The dealer
   then asks if you want another card by asking "Hit" or
   "Stand." If you choose to hit, the dealer gives you another
   card. If you choose to stand, the dealer draws or stands,
   and the game is played out according to the cards you and
   the dealer have. As with real Blackjack, the dealer stands
   on 17. The winner is announced only after both the player's
   and the dealer's hands are finished. */

   /*****************************************************************
   ANSI C standard header files appear next */
7  #include <stdio.h>
   #include <time.h>
   #include <ctype.h>
   #include <stdlib.h>

   /*****************************************************************
   Defined constants appear next */
7  #define BELL '\a'
   #define DEALER 0
   #define PLAYER 1

   /* Must keep two sets of totals for dealer and for player. The
   first set counts Aces as 1 and the second counts Aces as
   11. Unlike "real world" Blackjack, this program doesn't
```

```
   allow some Aces to be 1 while others Aces are 11 in the
   same hand. */
 7 #define ACELOW 0
   #define ACEHIGH 1
   /* Only one global variable is used in this entire program.
   The variable holds 0, which means false initially. Once the
   user enters his or her name in initCardsScreen(), this
   variable is set to 1 (for true), so the name is never asked
   for again the rest of the program. */
 5 int askedForName = 0;  /* False initially */
   /*************************************************************
   This program's specific prototypes */
   void dispTitle(void);
32 void initCardsScreen(int cards[52], int playerPoints[2],
   int dealerPoints[2], int total[2],
   int * numCards);
   int  dealCard(int * numCards, int cards[52]);
   void dispCard(int cardDrawn, int points[2]);
   void totalIt(int points[2], int total[2], int who);
   void dealerGetsCard(int *numCards, int cards[52],
   int dealerPoints[2]);
   void playerGetsCard(int *numCards, int cards[52],
   int playerPoints[2]);
   char getAns(char mesg[]);
   void findWinner(int total[2]);

   /*************************************************************
   C's program execution always begins at main() here */
 2 main()
   {
 5 int numCards;  /* Equals 52 at beginning of each game */
21 int cards[52], playerPoints[2], dealerPoints[2], total[2];
   char ans;  /* For user's Hit/Stand or Yes/No response */
14 do { initCardsScreen(cards, playerPoints, dealerPoints,
24 total, &numCards);
31 dealerGetsCard(&numCards, cards, dealerPoints);
 4 printf("\n");  /* Prints a blank line */
31 playerGetsCard(&numCards, cards, playerPoints);
   playerGetsCard(&numCards, cards, playerPoints);
14 do {
31 ans = getAns("Hit or stand (H/S)? ");
11 if (ans == 'H')
31 {  playerGetsCard(&numCards, cards,
   playerPoints);
   }
   } while (ans != 'S');
31 totalIt(playerPoints, total, PLAYER);
   /* Player's total */
14 do {
31 dealerGetsCard(&numCards, cards, dealerPoints);
   } while (dealerPoints[ACEHIGH] < 17);
   /* 17: Dealer stops */
   totalIt(dealerPoints, total, DEALER);
```

```
   /* Dealer's total */
   findWinner(total);
   ans = getAns("\nPlay again (Y/N)? ");
   } while (ans == 'Y');
32 return;
   }

 3 /*************************************************************
   This function initializes the face values of the deck of
   cards by putting four sets of 1-13 in the 52-card array. Also
   clears the screen and displays a title. */
30 void initCardsScreen(int cards[52], int playerPoints[2],
   int dealerPoints[2], int total[2],
   int *numCards)
   {
 5 int sub, val = 1;  /* This function's Work variables */
 6 char firstName[15];  /* Holds user's first name */
21 *numCards = 52;  /* Holds running total of
                       number of cards */
15 for (sub = 0; sub <= 51; sub++)  {  /* Counts from
                                         0 to 51 */
13 val = (val == 14) ? 1 : val;  /* If val is 14,
                                    reset to 1 */
21 cards[sub] = val;
13 val++;  }
15 for (sub = 0; sub <= 1; sub++)  /* Counts from 0 to 1 */
 9 { playerPoints[sub]=dealerPoints[sub]=total[sub]=0; }
31 dispTitle();
11 if (askedForName == 0)  /* Name asked for only once */
 4 { printf("\nWhat is your first name? ");
 8 scanf(" %s", firstName);
 5 askedForName = 1;  /* Don't ask prompt again */
 4 printf("Ok, %s, get ready for casino action!\n\n",
          firstName);
18 getchar();  /* Discards newline. You can safely */
   }             /* ignore compiler warning here.    */
32 return;
   }

 3 /*************************************************************
   This function gets a card for the player and updates the
   player's points. */
30 void playerGetsCard(int *numCards, int cards[52],
   int playerPoints[2])
   {
 5 int newCard;
31 newCard = dealCard(numCards, cards);
 4 printf("You draw: ");
31 dispCard(newCard, playerPoints);
   }

 3 /*************************************************************
   This function gets a card for the dealer and updates the
```

```
   dealer's points. */
30 void dealerGetsCard(int *numCards, int cards[52],
   int dealerPoints[2])
   {
 5 int newCard;
31 newCard = dealCard(numCards, cards);
 4 printf("The dealer draws: ");
31 dispCard(newCard, dealerPoints);
   }

 3 /*************************************************************
   This function gets a card from the deck and stores it in
   either the dealer's or the player's hold of cards. */
   int dealCard(int * numCards, int cards[52])
   {
 5 int cardDrawn, subDraw;
   time_t t;  /* Gets time for a random value */
20 srand(time(&t));  /* Seeds random-number generator */
   subDraw = (rand() % (*numCards));  /* From 0 to numcards */
   cardDrawn = cards[subDraw];
24 cards[subDraw] = cards[*numCards -1];  /* Puts top card */
24 (*numCards)-;                  /* in place of drawn one */
32 return cardDrawn;
   }

 3 /*************************************************************
   Displays the last-drawn card and updates points with it. */
30 void dispCard(int cardDrawn, int points[2])
   {
17 switch (cardDrawn) {
   case(11) : printf("%s\n", "Jack");
10 points[ACELOW] += 10;
21 points[ACEHIGH] += 10;
16 break;
   case(12) : printf("%s\n", "Queen");
10 points[ACELOW] += 10;
21 points[ACEHIGH] += 10;
16 break;
   case(13) : printf("%s\n", "King");
10 points[ACELOW] += 10;
21 points[ACEHIGH] += 10;
16 break;
   default :  points[ACELOW] += cardDrawn;
11 if (cardDrawn == 1)
   { printf("%s\n", "Ace");
10   points[ACEHIGH] += 11;
   }
   else
10 {  points[ACEHIGH] += cardDrawn;
 4    printf("%d\n", cardDrawn); }
   }
32 return;
   }
```

```
 3 /*****************************************************************
   Figures the total for player or dealer to see who won. This
   function takes into account the fact that Ace is either 1
   or 11. */
28 void totalIt(int points[2], int total[2], int who)
   {
 3 /* The following routine first looks to see if the total
   points counting Aces as 1 is equal to the total points
   counting Aces as 11. If so, or if the total points
   counting Aces as 11 is more than 21, the program uses
   the total with Aces as 1 only. */
11 if ((points[ACELOW] == points[ACEHIGH]) ||
   (points[ACEHIGH] > 21))
21 { total[who] = points[ACELOW]; }  /* Keeps all Aces
                                        as 1 */
   else
21 { total[who] = points[ACEHIGH]; }  /* Keeps all Aces
                                         as 11 */
11 if (who == PLAYER)  /* Determines the message printed */
 4 { printf("You have a total of %d\n\n", total[PLAYER]); }
   else
   { printf("The house stands with a total of %d\n\n",
            total[DEALER]); }
32 return;
   }

 3 /*****************************************************************
   Prints the winning player.  */
   void findWinner(int total[2])
   {
11 if (total[DEALER] == 21)
 4 { printf("The house wins.\n");
32   return;}
12 if ((total[DEALER] > 21) && (total[PLAYER] > 21))
 4 { printf("%s", "Nobody wins.\n");
32   return; }
12 if ((total[DEALER]>=total[PLAYER])&&(total[DEALER]<21))
 4 { printf("The house wins.\n");
32   return; }
12 if ((total[PLAYER] > 21) && (total[DEALER] < 21))
 4 { printf("The house wins.\n");
32   return; }
 4 printf("%s%c", "You win!\n", BELL);
32   return;
   }

 3 /*****************************************************************
   Gets the user's uppercase, single-character response. */
30 char getAns(char mesg[])
   {
 5 char ans;
 4 printf("%s", mesg);  /* Prints the prompt message passed */
18 ans = getchar();
```

```
   getchar();  /* Discards newline. You can safely */
               /* ignore compiler warning here.    */
19 return toupper(ans);
   }

 3 /*************************************************************
   Clears everything off the screen. */
30 void dispTitle(void)
   {
 5 int i = 0;
14 while (i < 25)     /* Clears screen by printing 25 blank */
 4 { printf("\n");    /* lines to "push off" stuff that     */
   i++; }           /* might be left over on the screen   */
                    /* before this program                */
 4 printf("\n\n*Step right up to the Blackjack tables*\n\n");
32 return;
   }
```

附录 C

ASCII 表

十 进 制	十六进制	二 进 制	ASCII字符
000	00	0000 0000	null
001	01	0000 0001	☺
002	02	0000 0010	●
003	03	0000 0011	♥
004	04	0000 0100	◆
005	05	0000 0101	♣
006	06	0000 0110	♠
007	07	0000 0111	●
008	08	0000 1000	■
009	09	0000 1001	○
010	0A	0000 1010	■
011	0B	0000 1011	♂
012	0C	0000 1100	♀
013	0D	0000 1101	♪
014	0E	0000 1110	♪♪
015	0F	0000 1111	☼
016	10	0001 0000	►
017	11	0001 0001	◄
018	12	0001 0010	↕
019	13	0001 0011	‼
020	14	0001 0100	¶
021	15	0001 0101	§
022	16	0001 0110	–
023	17	0001 0111	↨
024	18	0001 1000	↑
025	19	0001 1001	↓
026	1A	0001 1010	→
027	1B	0001 1011	←
028	1C	0001 1100	FS
029	1D	0001 1101	GS
030	1E	0001 1110	RS
031	1F	0001 1111	US
032	20	0010 0000	SP
033	21	0010 0001	!
034	22	0010 0010	"
035	23	0010 0011	#
036	24	0010 0100	$
037	25	0010 0101	%
038	26	0010 0110	&
039	27	0010 0111	'
040	28	0010 1000	(
041	29	0010 1001	)

（续）

十 进 制	十六进制	二 进 制	ASCII字符
042	2A	0010 1010	*
043	2B	0010 1011	+
044	2C	0010 1100	'
045	2D	0010 1101	-
046	2E	0010 1110	.
047	2F	0010 1111	/
048	30	0011 0000	0
049	31	0011 0001	1
050	32	0011 0010	2
051	33	0011 0011	3
052	34	0011 0100	4
053	35	0011 0101	5
054	36	0011 0110	6
055	37	0011 0111	7
056	38	0011 1000	8
057	39	0011 1001	9
058	3A	0011 1010	:
059	3B	0011 1011	;
060	3C	0011 1100	<
061	3D	0011 1101	=
062	3E	0011 1110	>
063	3F	0011 1111	?
064	40	0100 0000	@
065	41	0100 0001	A
066	42	0100 0010	B
067	43	0100 0011	C
068	44	0100 0100	D
069	45	0100 0101	E
070	46	0100 0110	F
071	47	0100 0111	G
072	48	0100 1000	H
073	49	0100 1001	I
074	4A	0100 1010	J
075	4B	0100 1011	K
076	4C	0100 1100	L
077	4D	0100 1101	M
078	4E	0100 1110	N
079	4F	0100 1111	O
080	50	0101 0000	P
081	51	0101 0001	Q
082	52	0101 0010	R

（续）

十 进 制	十六进制	二 进 制	ASCII字符
083	53	0101 0011	S
084	54	0101 0100	T
085	55	0101 0101	U
086	56	0101 0110	V
087	57	0101 0111	W
088	58	0101 1000	X
089	59	0101 1001	Y
090	5A	0101 1010	Z
091	5B	0101 1011	[
092	5C	0101 1100	\
093	5D	0101 1101	]
094	5E	0101 1110	^
095	5F	0101 1111	_
096	60	0110 0000	`
097	61	0110 0001	a
098	62	0110 0010	b
099	63	0110 0011	c
100	64	0110 0100	d
101	65	0110 0101	e
102	66	0110 0110	f
103	67	0110 0111	g
104	68	0110 1000	h
105	69	0110 1001	i
106	6A	0110 1010	j
107	6B	0110 1011	k
108	6C	0110 1100	l
109	6D	0110 1101	m
110	6E	0110 1110	n
111	6F	0110 1111	o
112	70	0111 0000	p
113	71	0111 0001	q
114	72	0111 0010	r
115	73	0111 0011	s
116	74	0111 0100	t
117	75	0111 0101	u
118	76	0111 0110	v
119	77	0111 0111	w
120	78	0111 1000	x
121	79	0111 1001	y
122	7A	0111 1010	z
123	7B	0111 1011	{

（续）

十 进 制	十六进制	二 进 制	ASCII字符
124	7C	0111 1100	¦
125	7D	0111 1101	}
126	7E	0111 1110	~
127	7F	0111 1111	DEL
128	80	1000 0000	Ç
129	81	1000 0001	ü
130	82	1000 0010	é
131	83	1000 0011	â
132	84	1000 0100	ä
133	85	1000 0101	à
134	86	1000 0110	å
135	87	1000 0111	ç
136	88	1000 1000	ê
137	89	1000 1001	ë
138	8A	1000 1010	è
139	8B	1000 1011	ï
140	8C	1000 1100	î
141	8D	1000 1101	ì
142	8E	1000 1110	Ä
143	8F	1000 1111	Å
144	90	1001 0000	É
145	91	1001 0001	æ
146	92	1001 0010	Æ
147	93	1001 0011	ô
148	94	1001 0100	ö
149	95	1001 0101	ò
150	96	1001 0110	û
151	97	1001 0111	ù
152	98	1001 1000	ÿ
153	99	1001 1001	Ö
154	9A	1001 1010	Ü
155	9B	1001 1011	¢
156	9C	1001 1100	£
157	9D	1001 1101	¥
158	9E	1001 1110	P_t
159	9F	1001 1111	*f*
160	A0	1010 0000	á
161	A1	1010 0001	í
162	A2	1010 0010	ó
163	A3	1010 0011	ú
164	A4	1010 0100	ñ

（续）

十 进 制	十六进制	二 进 制	ASCII字符
165	A5	1010 0101	Ñ
166	A6	1010 0110	ª
167	A7	1010 0111	º
168	A8	1010 1000	¿
169	A9	1010 1001	⌐
170	AA	1010 1010	¬
171	AB	1010 1011	½
172	AC	1010 1100	¼
173	AD	1010 1101	¡
174	AE	1010 1110	«
175	AF	10101111	
176	B0	1011 0000	▒
177	B1	1011 0001	▓
178	B2	1011 0010	▓
179	B3	1011 0011	│
180	B4	1011 0100	┤
181	B5	1011 0101	╡
182	B6	1011 0110	╢
183	B7	1011 0111	╖
184	B8	1011 1000	╕
185	B9	1011 1001	╣
186	BA	1011 1010	║
187	BB	1011 1011	╗
188	BC	1011 1100	╝
189	BD	1011 1101	╜
190	BE	1011 1110	╛
191	BF	1011 1111	┐
192	C0	1100 0000	└
193	C1	1100 0001	┴
194	C2	1100 0010	┬
195	C3	1100 0011	├
196	C4	1100 0100	─
197	C5	1100 0101	┼
198	C6	1100 0110	
199	C7	1100 0111	╟
200	C8	1100 1000	╚
201	C9	1100 1001	╔
202	CA	1100 1010	╩
203	CB	1100 1011	╦

（续）

十 进 制	十六进制	二 进 制	ASCII字符
204	CC	1100 1100	╠
205	CD	1100 1101	═
206	CE	1100 1110	╬
207	CF	1100 1111	╧
208	D0	1101 0000	╨
209	D1	1101 0001	╤
210	D2	1101 0010	╥
211	D3	1101 0011	╙
212	D4	1101 0100	╘
213	D5	1101 0101	╒
214	D6	1101 0110	╓
215	D7	1101 0111	╫
216	D8	1101 1000	╪
217	D9	1101 1001	┘
218	DA	1101 1010	┌
219	DB	1101 1011	█
220	DC	1101 1100	▄
221	DD	1101 1101	▌
222	DE	1101 1110	▐
223	DF	1101 1111	▀
224	E0	1110 0000	α
225	E1	1110 0001	ß
226	E2	1110 0010	Γ
227	E3	1110 0011	π
228	E4	1110 0100	Σ
229	E5	1110 0101	σ
230	E6	1110 0110	µ
231	E7	1110 0111	τ
232	E8	1110 1000	Φ
233	E9	1110 1001	Θ
234	EA	1110 1010	Ω
235	EB	1110 1011	δ
236	EC	1110 1100	∞
237	ED	1110 1101	ø
238	EE	1110 1110	∈
239	EF	1110 1111	∩
240	F0	1110 0000	≡
241	F1	1111 0001	±

（续）

十 进 制	十六进制	二 进 制	ASCII字符
242	F2	1111 0010	≥
243	F3	1111 0011	≤
244	F4	1111 0100	⌠
245	F5	1111 0101	⌡
246	F6	1111 0110	÷
247	F7	1111 0111	≈
248	F8	1111 1000	°
249	F9	1111 1001	∙
250	FA	1111 1010	·
251	FB	1111 1011	√
252	FC	1111 1100	n
253	FD	1111 1101	2
254	FE	1111 1110	■
255	FF	1111 1111	

本书用到的C函数和语句

名　称	说　明	所在章
break	结束循环	16
ceil()	上舍入某浮点数为最接近的整数（在MATH.H中定义）	20
continue	让循环从头开始	16
do-while	与while类似	14
default	switch中默认的分支	17
else	与if语句一起使用。如果条件为假，则执行其他代码	11
exit()	退出程序（在STDLIB.H中定义）	17
fabs()	返回绝对值（在MATH.H中定义）	20
fclose()	关闭文件（在STDIO.H中定义）	28
feof()	检查文件结尾（在STDIO.H中定义）	28
fgetc()	从文件中读取一个字符（在STDIO.H中定义）	29
fgets()	从文件中读取字符串（在STDIO.H中定义）	28
floor()	下舍入某浮点数为最接近的整数（在MATH.H中定义）	20
fopen()	打开一个文件（在STDIO.H中定义）	28
for	让代码重复执行一定次数的循环	15
fprintf()	输出到文件中（在STDIO.H中定义）	28
fputc()	把一个字符写到文件中（在STDIO.H中定义）	29
free()	释放堆内存（在STDLIB.H中定义）	26
fseek()	定位到文件的指定部分（在STDIO.H中定义）	29
getch()	从键盘读取一个、不缓存的字符（在STDIO.H中定义）	18
getchar()	从键盘读取一个、缓存的字符（在STDIO.H中定义）	18
gets()	从键盘读取一个字符串（在STRING.H中定义）	19
if	测试条件是否是真。如果为真，就执行相应的代码	11
isalpha()	测试一个字符是否是字母（在CTYPE.H中定义）	19
isdigit()	测试一个字符是否是数字（在CTYPE.H中定义）	19
islower()	测试一个字符是否是小写字母（在CTYPE.H中定义）	19
isupper()	测试一个字符是否是大写字母（在CTYPE.H中定义）	19
malloc()	分配堆内存（在STDLIB.H中定义）	26
pow()	幂运算（在MATH.H中定义）	20
printf()	输出到屏幕（在STDIO.H中定义）	4
putchar()	输出一个字符到屏幕（在STDIO.H中定义）	18
puts()	输出一个字符串到屏幕（在STRING.H中定义）	19
rand()	返回一个随机数（在STDLIB.H中定义）	20
return	从函数返回一个值	32
scanf()	从用户获取数据（在STDIO.H中定义）	8
sqrt()	返回平方根（在MATH.H中定义）	20
srand()	为rand()函数设置种子（在STDLIB.H中定义）	20
strcat()	连接两个字符串（在STRING.H中定义）	19
strcpy()	把一个字符串赋给一个变量（在STRING.H中定义）	6
switch	根据测试条件，可以跳到不同的代码块	17
tolower()	把一个字母转化成小写（在CTYPE.H中定义）	19
toupper()	把一个字母转化成大写（在CTYPE.H中定义）	19
while	只要条件为真，就重复执行其中的代码	14

复合赋值运算符

复合运算符	示例	等价语句
*=	total *= 1.25;	total = total * 1.25;
/=	amt /= factor;	amt = amt / factor;
%=	days %= 3;	days = days % 3;
+=	count +=1;	count = count + 1;
-=	adjust -= 0.5;	adjust = adjust - 0.5;

运算符优先级表

级别	运算符	结合性
1	()（括号）、[]（数组元素）、.（结构成员引用）	从左到右
2	-（负号）、++（自增）、—（自减）、&（取地址）、*（指向）、sizeof()、!（非运算符）	从右到左
3	*（乘号）、/（除号）、%（模）	从左到右
4	+（加号）、-（减号）	从左到右
5	<（小于）、<=（小于等于）、>（大于）、>=（大于等于）	从左到右
6	==（等于）、!=（不等于）	从左到右
7	&&（逻辑与）	从左到右
8	\|\|（逻辑或）	从左到右
9	? :（条件运算符）	从右到左
10	=、*=、/=、%=、+=、-=	从右到左
11	,（逗号运算符）	从左到右

基本的fopen()模式字符串

模式	说明
"w"	写（write）模式，无论文件是否存在都会创建一个新文件
"r"	读（read）模式，允许你读一个已经存在的文件。如果文件不存在，就会产生一个错误
"a"	附加（append）模式，允许添加数据到文件的末尾，或者当文件不存在时创建文件

随机访问的fopen()模式

模式	说明
"r+"	打开一个已经存在的文件以供读和写
"w+"	打开一个新文件以供读和写
"a+"	以附加模式打开一个文件（文件指针指向文件的结尾），但允许你回到文件的前面进行读和写

转化字符

转化字符	说明
%d	整数
%f	浮点数
%c	字符
%s	字符串

转义序列

代码	说明
\n	新行
\a	警报（电脑响铃）
\t	制表符
\\	反斜杠
\"	引号

逻辑运算符

逻辑运算符	含义
&&	与
\|\|	或
!	非

C的关系运算符

关系运算符	说明
==	等于
>	大于
<	小于
>=	大于等于
<=	小于等于
!=	不等于

指针运算符

运算符	说明
&	取址运算符
*	取值运算符

fseek()中出现的orgin的值

origin	说明
SEEK_SET	文件的开头
SEEK_CUR	文件的当前位置
SEEK_END	文件的结尾